Informatik I – Exercise 1

Working with UNIX

(mini-project: file manipulation/processing/visualization using UNIX)

Document version: 15.09.2023

Exercise week 1:	25.09.2023-01.10.2023
Exercise week 2:	02.10.2023-08.10.2023
Deadline for the report:	09.10.2023

Summary

The goal of this exercise is to perform your initiation to the UNIX operating system, which you will use for all subsequent exercises of this course and - very likely - often later in your studies at ETH and in your scientific career. This includes: (*i*) opening and using a UNIX terminal; (*ii*) understanding the structure of the UNIX filesystem; (*iii*) knowing how to navigate the filesystem and perform basic file operations; (*iv*) being introduced to some more advanced UNIX commands; (*v*) learning how to use a few applications that are necessary for the following exercises. In the sandbox part (week 1), you will get acquainted with the new concepts by experimenting with simple examples. In the mini-project part (week 2), you will use what you just learned for performing a series of simple tasks (file manipulation/processing and data visualization) using UNIX. In the lecture script booklet, the material covered by this exercise corresponds to lecture slides LN@000 to LN@022.

1.1 Week 1 - Sandbox

1.1.1 Before starting

First, make sure you have read carefully Ex. 0, the *starting document for students*, which explains how to use this exercise script (Secs. 0.2 and 0.3). You can ignore the parts describing how to start your exercise (Sec. 0.5) and how to return your solution for the mini-project (Sec. 0.8). Since this is the first exercise, we will perform these operations in the course of the exercise[1]. On the other hand, keep very much in mind the useful tip of Sec. 0.10. Whenever you encounter a specific computational problem or you don't remember exactly how to do something (here, related to UNIX), trying to find a solution yourself using internet (search engine with a good combination of keywords) is nowadays an essential component of computational skills[2].

Note that there is a lot of ground to cover in the sandbox part, whereas the mini-project part is relatively easy (at least tasks A and B). Your assistant may thus suggest you to leave some of the week 1 material for week 2.

1.1.2 Logging in

To gain access into the computers of our computer rooms HCI D 267 or HIT F21, you first need to *log in*. This is done by entering your `nethz` user name and password[3]. Please respect (strictly!) three basic rules: (*i*) *never tell your password* to anyone; (*ii*) *always log out of the machine* when you leave the room[4]; (*iii*) *never turn off a computer* using its on/off switch or (worse!) by

[1] But these sections will be essential for the next exercises, from Ex. 2 onward.

[2] This applies even (and maybe especially) to the professionals!

[3] These are the same user name and password you use for your ETHZ e-mail account.

[4] This way, you make sure that no one else can access your account, and possibly mess up with it.

pulling the plug[5]. Note that the user account you will access is not local to the computer you use right now, it is your *ETHZ-student account*. This account is stored centrally by the ETHZ Informatikdienste, and exported to your current computer over the network. This means that the same material is accessible at any time from any other computer in and out of ETH. Also, this account will remain yours even after you complete the Informatik I course. With this in mind, you can now log into the machine in front of you[6].

1.1.3 UNIX and GUI

`UNIX` is an operating system that started being developed in the seventies. The more recent versions are commonly referred to as `LINUX`, and there are many variants of them[7]. They represent a modern `UNIX` alternative to the operating systems Windows (Microsoft) and MacOS (Apple). Nowadays, `LINUX` (and not Windows or MacOS) is the system most commonly used for IT applications. It is also routinely used by all academic/industrial scientific research groups with a significant theoretical/computational component, which is why you have to learn how to use it.

The early `UNIX` computers of the seventies had only one plain-text terminal[8]. However, the modern `LINUX` distributions come with a *graphical user interface* (GUI), which makes them look at first sight very similar to a Window or MacOS PC. The computers in our computer rooms HCI D 267 and HIT F21 run a `LINUX` distribution called "Fedora". After you log in, you will see that its GUI makes it feel very much like a PC or a Mac. And indeed, you can do essentially the same things as on a PC or a Mac using menus/windows and click/drag/drop operations with the mouse. Where you will find (and use!) the `UNIX` operating system is inside a specific type of windows called *terminal* windows. And we will open one right now...

1.1.4 Terminal window

Once you are logged in, open the "file manager" (see here[9] how to do that). This corresponds to the "windows explorer" of Windows or the "finder" of MacOS. You should see folders with the names `Desktop`, `Documents`, `Downloads`, `homepage`, `Music`, `Pictures`, `profiles`, `private`, `Templates` and `Videos`. These are names you are probably familiar with from working with your own computer. Create a new folder called `ex1` in the usual way, by right-clicking into the file-manager window, selecting "new folder", and entering the desired name `ex1`. For a while, we will keep the file-manager window open, and we will use it to compare the `UNIX` way of doing things to the PC/Mac way. And now you can open a terminal window (see here[10] how to do that): we will start with some `UNIX`.

When the terminal window opens, it is almost empty, containing something like

```
bash-4.4$
```

[5] If your computer seems "frozen", talk to your assistant.

[6] And let's bet that you already did!

[7] For example, "Red Hat", "Fedora", "Cent OS", "Ubuntu", "Debian", and many others...

[8] What you would nowadays call one single window, and with only ASCII text in it!

[9] Click on "activities" in the top left corner of the screen. The vertical set of icons that pops up at the left of the screen lists your "favorite applications". Click on the icon representing a file cabinet with drawers, this is the "file manager".

[10] Click on "activities" in the top left corner of the screen, and select "show applications" (or, in one step, press the `<WINDOWS>` key on the keyboard). In the search bar that appears at the top, type "terminal", and you will see the corresponding icon appear. Since you will often use terminal windows, drag the icon to the left of the screen to add it to your "favorite applications". You can exit the search by pressing the `<ESC>` key. To open the terminal, just double-click on its icon.

This is your *command prompt*, *i.e.* a text from the computer telling you that it is your turn to write something in the window. Type the command `csh` and press the `<ENTER>` key[11]. You should now see in the window something like

```
bash-4.4$ csh
[user@comp ~]$
```

The second line is again a command prompt (your turn to write!). But it has changed a bit in appearance, now providing some more information, namely your user name (that we will systematically write `user` in this document, since we don't know your name), the name of the computer you are using (that we will systematically write `comp`, since we also don't know what computer you chose to work on), and the directory (folder[12]) you are currently in. The user name is your `nethz` user name, the one you just used for logging in. The computer name will be some (not very glamorous) name that was selected by ETHZ Informatikdienste for labelling the specific computer in the specific room. This name is actually written (sticker) on the computer itself. Finally, the directory will read ~, indicating that you are now in your *home directory* (see next section).

And before you start typing a bunch `UNIX` commands, here is a little survival trick. Whenever you type a command, it is executed, and then you get the prompt back for typing a new command. If it happens that a command *hangs (freezes)* or *keeps spitting out (useless) data* for ever[13], without giving you back the prompt, don't panic! Just type[14] `<CTRL-c>` in the terminal window. This should interrupt the process immediately[15] and you will get your prompt back.

Please also keep two *important points* in mind: First, almost everything in `UNIX` is *case sensitive*, *i.e.* '`A`' is *not* the same thing as '`a`'. This applies to commands, file names, directory names, ... So, always make sure you use the right case! Second, different keywords *must be separated by spaces*. It does not matter how many, but you need at least one. For example, we will see in a moment that the command `ls ~/ex1` lists the content of subdirectory `ex1` in your home directory. Typing `ls ~`/ex1 (multiple spaces) will work. But typing `ls`~/ex1 (no space) will not (error message)! And typing `ls ~ /ex1` will work, but give a different result as `UNIX` will see three keywords instead of two (this will list the contents of both your home directory and the directory `/ex1` under the root directory, which is probably not what you want). You don't need to understand these commands yet. But you have to keep in mind already now that space or no-space makes a difference!

1.1.5 The `UNIX` filesystem

The structure of the `UNIX` *filesystem* has been explained at the lecture (LN@003 and LN@005) and only the essential information is recalled here, see Fig. 1.1.

The top-most directory is called the *root directory*. It contains everything. Starting from this one, directories and files are organized in the form of an upside down tree. In fact, this structure is very much the same as in Windows or MacOS. The main difference is that you do not manipulate

[11] This tells the computer that we will use the *c-shell*. Shells are "flavors" of `UNIX` in which some commands work (slightly) differently. You don't need to know about these. Here, we use the c-shell variant (instead of the default, which is `bash`) as the most convenient one for the purpose of these exercises. If you once get bored of typing `csh` everytime you open an new terminal window, have a look at Sec. 1.1.14 where we tell you how to bypass this step.

[12] The words "folder" and "directory" are synonyms. In `UNIX`, one uses more often the word "directory", which we will normally do throughout this script.

[13] Note that if this happens, this will normally be your fault and not that of the computer. Usually, `UNIX` systems only go banana when you give them silly instructions (there are exceptions, of course, but they are relatively uncommon). Can you say the same from your Windows or MacOS PC?

[14] That is, press simultaneously the `<CTRL>` key and the letter `<c>`.

[15] For real die-hard processes, `<CTRL-c>` may sometimes fail to work. Then try `<CTRL-z>`, which brings the process to sleep. Sometimes a morphine shot does more damage than a bullet.

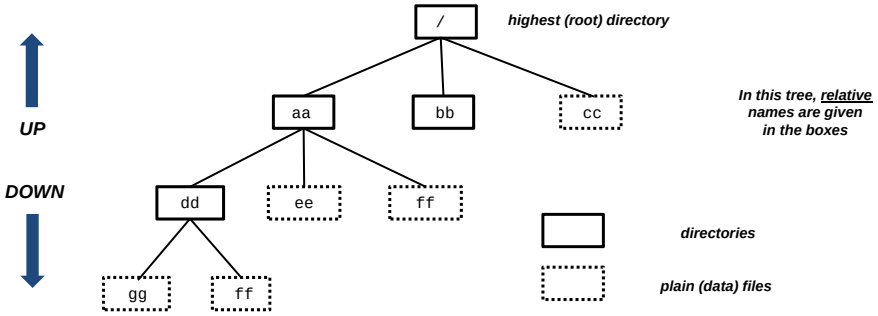

Figure 1.1: The UNIX filesystem.

this tree using a number of separate file-manager windows (or "windows explorer" or "finder") along with click/drag/drop operations. Instead you do everything in a single terminal window by means of *text commands*. To replace the different windows of Windows or MacOS showing the contents of different folders at the same time on your desktop, the single terminal is associated with a single *current directory* (also called the *working directory*). It is the directory "you are in" at the moment.

There are two ways to refer to directories and files in the filesystem. You can use an *absolute path* from the root directory, which is indicated by starting the path with a `/` (in Fig. 1.1, an example of absolute path for a file is `/aa/dd/ff`). Or, given a current directory, you can use a *relative path* (*i.e.* relative to the current directory), which is indicated by *not* starting with a `/` (in Fig. 1.1, if your current directory is `/aa`, the absolute path for the same file is `dd/ff`).

At the moment, your terminal window has your *home directory*[16] as current directory, which is indicated by the symbol ~ in the command prompt. Unlike on a PC or a Mac, the terminal does not automatically show the contents of the current directory. To see it, we have to use our very first[17] `UNIX` command `ls` (shorthand for "list"). Just type[18] `ls` and press `<ENTER>`. You should see this[19]

```
[user@comp ~]$ ls
Desktop    Downloads   homepage  Pictures   profiles   Templates
Documents  ex1         Music     private    Public     Videos
```

The command `ls` has printed in the terminal window a list with the contents of the current directory, which is for the moment your home directory. What you see are the same directories and files you have seen earlier in the file-manager window, including the folder `ex1` that we just created in Sec. 1.1.4. Now is time to learn how to navigate the filesystem and perform basic file operations.

1.1.6 Filesystem navigation

As any good sailor will tell you (even maybe a bad one): the first thing to do if you want to navigate to some far-away destination is... to figure out where you are now. In `UNIX` this is done

[16] Your home directory is the directory you "own" in the ETHZ account system as a specific ETHZ-student, *i.e.* it is the top directory of your ETHZ-student account (but not that of the whole system, which is `/`).

[17] Well, if we are strict, your very first `UNIX` command was actually `csh`.

[18] Note that it is `ls` (first letter is a lower case L) and not `1s` (first letter is *not* a one!). Like most `UNIX` commands, the command name is chosen to be short (quick to type!) but still intuitive (analogy to the word "list").

[19] After this list, you get the prompt back, which we will not always show explicitly in the listings of this exercise.

using the command `pwd` (shorthand for "print working directory") in the terminal window. Type `pwd` and you should get

```
[user@comp ~]$ pwd
/home/user
```

The terminal has printed the absolute path of your current directory, which is `/home/user`. In the prompt, it is written as ∼, which is an exact synonym for this absolute path. Thus, you are within the root directory `/` (obviously, as everything is!), within a subdirectory of it called[20] `home` (which itself contains home directories of ETHZ users[21]), and in a subdirectory of the latter called `user` (*i.e.* named with your `nethz` user name and belonging to you).

Now let's change the current directory of the terminal window to the subdirectory `ex1` that we created earlier in your home. In the file-manager window, do that by double-clicking on the icon for the folder `ex1`. In the terminal window, do that by using the command `cd` (shorthand for "change current directory"), followed by the path (absolute or relative) of the subdirectory you want to change to, namely `ex1` (*i.e.* here, a relative path).

```
[user@comp ~]$ cd ex1
[user@comp ~/ex1]$
```

You will notice that the command prompt has changed appropriately. Instead of ∼, it now indicates ∼`/ex1` as your current directory.

At this point, you can run a small experiment (first guess the outcome and write it on a piece of paper, and only then try it out on the computer!). What will happen if you type in sequence: (*i*) `pwd`; (*ii*) `ls`; (*iii*) `cd /home/user/ex1` (recall that `user` stands for your `nethz` user name); (*iv*) `pwd`. If you guessed wrongly, explanations can be found here[22].

If the emptiness of the current directory is making you sad, we will remedy this situation by creating a new subdirectory in there named `my_dir`. In the terminal window, do that by using the command `mkdir` (shorthand for "make directory"), followed by the path (absolute or relative) of the directory you want to create, namely `my_dir` (*i.e.* here, a relative path).

```
[user@comp ~/ex1]$ mkdir my_dir
```

Now check the file-manager window, where a small miracle[23] should have happened: an icon has just appeared for the new folder. Question: would the command `mkdir /home/user/ex1/my_dir` have worked as well? The answer is here[24]

At this point, you can again run a small experiment (first guess the outcome, and only then try it out!). What will happen if you type in sequence: (*i*) `pwd`; (*ii*) `ls`; (*iii*) `cd my_dir`; (*iv*) `pwd`. If you guessed wrongly, explanations can be found here[25]. To keep the file-manager window synchronized with the terminal window, double-click on the folder `my_dir` in this window.

[20] Note that the directory `home` is not *your* home directory, which is `/home/user`. In fact, this directory should better be called `homes` (plural), because it contains the home directories of all users. But calling it `home` is (unfortunately) the usual practice in `UNIX`-based IT.

[21] As will be mentioned later, only the home directories of users that recently worked on this computer (*i.e.* since its last reboot) will actually be there.

[22] The command `pwd` will print the absolute path of your current directory, which is now `/home/user/ex1`. The command `ls` will list the contents of your current directory, *i.e.* nothing as it is for the moment empty. The command `cd /home/user/ex1` will have no effect, as `/home/user/ex1` is already your current directory This illustrates that `cd` works equally well with an absolute path as with a relative one. And obviously, the last command `pwd` will print the same as before, `/home/user/ex1`.

[23] It is *really* a miracle, since you created this new folder directly using `UNIX` and without a single mouse click!

[24] Yes, this would have created the same directory, but using an absolute instead of a relative path.

[25] The command `pwd` will print the absolute path of your current directory, which is still `/home/user/ex1`. The command `ls` will list the contents of your current directory. For the moment, there is only one object listed, the new directory `my_dir` that you just created. The command `cd my_dir` will change the current directory to the

In the previous steps, we have learned how to use `cd` to move *down* the tree using relative paths (from your home, to the subdirectory `ex1` in there, and then to the subdirectory `my_dir` in there). You can also easily go *up* one folder with the relative path name `..`, which means "one level up". In the terminal window, enter the command `cd ..` and then type `pwd`

```
[user@comp my_dir]$ cd ..
[user@comp ~/ex1]$ pwd
/home/user/ex1
```

All correct! You are back with your current directory set to `/home/user/ex1`. To keep the file-manager window synchronized with the terminal window, click on the button marked "ex1" in this window, which will take it back to the `ex1` directory. There is another convention we should mention, namely the relative path name `.`, which means "in the current directory". For the moment, it is not very useful to you[26] but we will see two examples later (Secs. 1.1.7 and 1.1.9.4.

All `UNIX` commands that refer to files or directories (like `cd` and `mkdir` above) will accept both absolute and relative paths. With a bit of experience, you will find out for yourself when it is more convenient (*i.e.* quicker) to use one or the other. As seen above, in terms of relative paths, there are two useful shortcuts: (*i*) `.` for "in the current directory"; (*ii*) `..` for "in the parent directory", *i.e.* "one level up". In terms of absolute paths, there are also useful shortcuts: (*i*) `/` stands for the root directory; (*ii*) ~ stands for your home directory; (*iii*) ~`some_user` stands for the home directory of a given user; (*iv*) the command `cd` alone (no path specified) is equivalent to `cd` ~. The last one is very useful if you get lost in the filesystem. Just type `cd` alone, and you are brought back home. Finally, it is possible to use multiple-step paths. For example, `cd ../..` will take you two steps up, and `cd /home/user/./ex1/..` is (a complicated way[27] to) take you home.

Before going on, make sure that you have understood *very clearly* the operation of the command `cd`, and in particular the difference between typing `cd` (no argument → take me to my home directory) and `cd ..` (with argument "dot-dot" → take me to the parent directory, *i.e.* one level up). Also, have a look (now!) at Tab. 1.1, where the three first blocks summarize what we have seen until now about `csh`, `pwd` and `cd`.

From this point onward, we will stop referring all the time to the file-manager window (just occasionally). If you wish, you can keep it open and use it from time to time to check that what you see/do in the `UNIX` terminal window matches what you would expect from Windows or MacOS.

And now is time to navigate a bit on your own. For this, try the following things (if you don't remember how to, then read the footnotes). First, go to the root directory (using an absolute path), check that you are at the right place, and list the contents of the directory[28]. Here, you will find a number of subdirectories that are essential for the system, as well as the directory `home` which contains the home directories of all users. At this location (and in all the subdirectories except yours, `/home/user`), you do not have the permission to change anything. This privilege is reserved to the operating system and the system administrators. Second, still in root, try to go one

---

subdirectory `my_dir`. Note that we used again `cd` with a relative path name, but we could do this as well with an absolute one, as `cd /home/user/ex1/my_dir`. This change will also cause the prompt to change. Logically, it should read ~`/ex1/my_dir`, but when you are more than one level down from your home, it is simplified, to avoid having a too long prompt. Finally, the command `pwd` will print the absolute path of your new current directory, which is now `/home/user/ex1/my_dir`.

[26] For example, `cd .` means "stay where you are" and `mkdir ./dir` will do the same as `mkdir dir`. We don't have time to waste on such a useless symbol, it seems, but wait a bit...

[27] The single dot in the middle is particularly silly, as it just says that we "stay where we are" before doing the next change.

[28] The commands are `cd /`, then `pwd`, and then `ls`.

level up (using a relative path) and check where you are[29]. You will see that the `cd` command did not do anything, you are still in the root directory. This is logical as it is the top-most directory. There is nothing higher (except, maybe, the sky). Third, go to `/home` (using a relative path), check that you are at the right place, and list the contents of the directory[30]. Here, you would expect to find the home directories of all ETHZ users including yours. In practice, it is likely that you will only find yours. Look here[31] if you want to know why. Fourth, go to `/home/user/ex1/my_dir` (using an absolute path) and check that you are at the right place[32]. Fifth, go back to the root directory (now using a relative path) and check that you are at the right place[33]. Finally, go back home in the simplest possible way[34].

1.1.7 Basic file operations

Enough navigation - we don't want to make you seasick already at this first exercise! Let's now play a bit with files. Go to the directory `~/ex1` (by now, you should know how to) and type `touch apple.txt`, then `ls`

```
[user@comp ~]$ cd ~/ex1
[user@comp ~/ex1]$ touch apple.txt
[user@comp ~/ex1]$ ls
my_dir apple.txt
```

The command `touch` creates an empty file with the given name[35]. Note that the output of `ls` is colorful. On many `UNIX` systems (like the one you use now), the files and directories are listed in different colors, which makes things easier to distinguish[36].

Recall that `UNIX` commands that operate on files or directories also work when you use more complicated (absolute or relative) path names. For example, go to your home directory and type `touch ex1/pear.txt`, then `ls ex1`

```
[user@comp ~/ex1]$ cd
[user@comp ~]$ touch ex1/pear.txt
[user@comp ~]$ ls ex1
my_dir apple.txt pear.txt
```

Here, `touch` has created a file in the indicated directory `ex1` instead of the current one, and `ls` with the name of a directory has listed the contents of this directory instead of the current one. If you wish, you can check in the file-manager window that the directory contents of `~/ex1` is indeed as listed.

[29] The commands are `cd ..` and then `pwd`.

[30] The commands are `cd home`, then `pwd`, and then `ls`.

[31] The home directory of a user is only linked (one says "auto-mounted") over the network when the user actually works on the specific computer, and the link only remains there until the next reboot of the machine. So, for week 1 of Ex. 1, what you see here will depend a lot whether you are in a Tuesday or a Friday group. If it happens that there is someone else listed, say the student `batman`, you can try to go to his home using `cd ~batman`. Besides the ethical conflict that may arise in your own brain, this will result in an error message as you don't have the permission to spy on other students (so, `UNIX` implicitly prevents ethical conflicts).

[32] The commands are `cd /home/user/ex1/my_dir` and then `pwd`.

[33] The commands are `cd ../../../..` (we have to go four levels up) and then `pwd`.

[34] The command is `cd`.

[35] This command has another function when the file already exists. In this case, it leaves it unchanged but updates its last-modification date/time, thus the name "touch".

[36] Actually, as we will see briefly in Sec. 1.1.9.4, `ls` does not print in color by default. But this command has been aliased for you to `ls --color=auto`. On systems that do not show colors, you can also use `ls -p` to distinguish between files and directories. The `-p` option will add a trailing `/` at the end of directory (but not file) names. Feel free to try.

Now we can turn to the basic commands that serve to copy, rename, relocate and delete files or directories. These commands are essential and you should know/master them if you do not want to keep struggling with `UNIX` instead of learning `C++` in the next exercises.

To *copy* a file you can use the command `cp` (shorthand for "copy"). This command usually has two arguments, source and destination. The source is the path of the file you want to copy. The destination is either the path of the copy (including a chosen filename), or the name of a directory where you want to place a copy with the same filename. Watch out that if the destination is a file that already exists, it will be mercilessly overwritten (original contents lost) when copying. For example, relative to the current directory `~/ex1`, the commands below will: (*i*) make a copy of the file `apple.txt` that is called `orange.txt`; (*ii*) make a copy of the file `apple.txt` with the same name in the subdirectory `my_dir`; (*iii*) make a copy of the file `apple.txt` with the name `ananas.txt` in the subdirectory `my_dir`; (*iv*) make a copy of the newly created file `my_dir/ananas.txt` with the same name in the current (`~/ex1`) directory

```
[user@comp ~]$ cd ~/ex1
[user@comp ~/ex1]$ cp apple.txt orange.txt
[user@comp ~/ex1]$ cp apple.txt my_dir
[user@comp ~/ex1]$ cp apple.txt my_dir/ananas.txt
[user@comp ~/ex1]$ cd my_dir
[user@comp my_dir]$ cp ananas.txt ..
[user@comp my_dir]$ cd ..
[user@comp ~/ex1]$
```

Here, we recall that `..` means "one level up". Now, try to guess what the commands `ls` and `ls my_dir` will print (if needed, make a little sketch on paper, because these are lots of fruits to keep track of in your head), and then verify that the result is as you expect[37]. If you wish, you can also check all this in the file-manager window[38].

Three important remarks on `cp`. First, the behavior of `cp` depends on the second argument of the command. If the destination has a name that does not yet exist, a file with the given name will be created for the copy. If it already exists and is a directory, a file with an unchanged name will be created in the indicated directory for the copy. And if it already exists and is a file, the existing file will be overwritten for the copy. Second, `cp` will only copy files and it will stubbornly refuse to copy directories (giving you an angry warning message about this). You are encouraged to check this by yourself, just try `cp my_dir copy_of_my_dir`. To copy directories (including their full content), you have to use[39] `cp -r`. Let's give it a try

```
[user@comp ~/ex1]$ cp -r my_dir copy_of_my_dir
[user@comp ~/ex1]$ ls
copy_of_my_dir my_dir ananas.txt apple.txt orange.txt pear.txt
[user@comp ~/ex1]$ ls copy_of_my_dir
ananas.txt apple.txt
```

It has worked! Third, it is allowed to copy more than one file to a directory in one command. For example, `cp file1 file2 dir` will copy both files (under their original names) to the given directory.

[37] The command `ls` should print the contents of `~/ex1`, namely `my_dir`, `ananas.txt`, `apple.txt`, `orange.txt` and `pear.txt`. The command `ls my_dir` should print the contents of `~/ex1/my_dir`, namely `ananas.txt` and `apple.txt`.

[38] In the file-manager window, you can also copy files. For example, for the `cp apple.txt my_dir/ananas.txt`, you would have to: go to the `ex1` directory, copy the file to the clipboard (`<CTRL-c>`), go to the `my_dir` subdirectory, paste the file from the clipboard (`<CTRL-v>`), and then rename the file. These are many more manipulations than with `UNIX`!

[39] The option `-r` stands for "recursive", which mean that the command applies to the object, to what is inside, to what is inside the inside, and so on.

To *rename* or *relocate* files or directories you can use the command `mv` (shorthand for "move"). Unlike `cp`, the command `mv` will remove the original version of the object (the one under the old name or at the old location). This command usually has two arguments, source and destination. The source is the path of the file/directory you want to rename or relocate. The destination is either the new path desired for the file/directory (including a chosen file/directory name), or the name of a directory you want to move the file/directory into with the same name. Here again, watch out that if the source is a file and the destination is a file that already exists, the latter will be pitilessly overwritten (original copy lost) when doing the move. For example, relative to the current directory `~/ex1`, the commands below will: (*i*) rename the file `apple.txt` to `mango.txt`; (*ii*) relocate the file `pear.txt` to the subdirectory `my_dir`; (*iii*) rename/relocate the file `orange.txt` to `banana.txt` in the subdirectory `my_dir`; (*iv*) relocate the file `my_dir/apple.txt` to the current (`~/ex1`) directory.

```
[user@comp ~/ex1]$ mv apple.txt mango.txt
[user@comp ~/ex1]$ mv pear.txt my_dir
[user@comp ~/ex1]$ mv orange.txt my_dir/banana.txt
[user@comp ~/ex1]$ mv my_dir/apple.txt .
```

Here, we recall that `.` means[40] "in the current directory". Now, try to guess that the commands `ls` and `ls my_dir` will print (again, maybe make a little sketch on paper), and then verify your that the result is as you expect[41]. If you wish, you can also check all this in the file-manager window[42].

Three important remarks on `mv`. First, like `cp`, the behavior of `mv` depends on the second argument of the command. If the destination has a name that does not yet exist, a file/directory with the given name will be created for the renamed source. If it already exists and is a directory, a file/directory with an unchanged name will be created in the indicated directory for the relocated source. And if it already exists and is a file, the existing file will be overwritten for the renamed/relocated source. Second, unlike `cp`, the command `mv` can be used to rename/relocate directories (including their full content). Let's give it a try

```
[user@comp ~/ex1]$ mv copy_of_my_dir my_dir
[user@comp ~/ex1]$ ls
my_dir ananas.txt apple.txt mango.txt
[user@comp ~/ex1]$ ls my_dir
copy_of_my_dir ananas.txt banana.txt pear.txt
```

It has worked! Third, it is allowed to relocate more than one file/directory to a directory in one command. For example, `mv file1 file2 dir1 dir2` will relocate both files as well as the first directory (under their original names) into the second directory.

We have generated a nice basket of fruits... time to clean up a bit. To *delete* files you can use the command `rm` (shorthand for "remove"). This command usually has one argument, the target. The target is the path of the file you want to delete. Here, watch out that the deletion of a file

[40] This is one example where the symbol `.` is useful (in Sec. 1.1.6, we promised you we would give two!).

[41] The command `ls` should print the contents of `~/ex1`, namely `copy_of_my_dir`, `my_dir`, `ananas.txt`, `apple.txt` and `mango.txt` (note that `apple.txt` is not the file that was initially in `ex1`, as this file was renamed to `mango.txt`; it is the `apple.txt` that was moved from `my_dir` by the last `mv` command). The command `ls my_dir` should print the contents of `~/ex1/my_dir`, namely `ananas.txt`, `banana.txt` and `pear.txt`.

[42] In the file-manager window, you can rename files by directly changing their names. And you can also relocate files. For example, for the `mv orange.txt my_dir/banana.txt`, you would have to: go to the `ex1` directory, cut the file to the clipboard (`<CTRL-x>`), go to the `my_dir` subdirectory, paste the file from the clipboard (`CTRL-v`), and then rename the file. Here again, the `UNIX` way is significantly faster!

is irreversible[43] (the same applies if you overwrite a file with a clumsy `cp` of `mv`). For example, relative to the current directory ∼`/ex1`, the commands below will: (*i*) delete the file `apple.txt`; (*ii*) delete the file `my_dir/pear.txt`.

```
[user@comp ∼/ex1]$ rm apple.txt
[user@comp ∼/ex1]$ rm my_dir/pear.txt
```

Now, try to guess that the commands `ls` and `ls my_dir` will print (now you know the drill: little sketch on paper, *etc...*), and then verify that the result is as you expect[44]. If you wish, you can also check all this in the file-manager window[45].

Two important remarks on `rm`. First, `rm` will only delete files and it will obdurately refuse to delete directories (giving you a crotchety warning message about this)[46]. You are encouraged to check this by yourself, just try `rm my_dir`. To remove empty directories you have to use the command `rmdir` instead. This command will also refuse to delete non-empty directories (also giving you a warning message about this). To remove non-empty directories (including their full content), you have to use[47] `rm -r`. Let's give it a try

```
[user@comp ∼/ex1]$ rm -r my_dir/copy_of_my_dir
[user@comp ∼/ex1]$ ls my_dir
ananas.txt banana.txt
```

It has worked! Keep in mind, though, that the `rm -r` command should be *used with great caution*. It can be truly devastating, by quickly and irreversibly deleting a massive amount of files. Second remark on `rm`, it is allowed to delete more than one file in one command. For example, `rm file1 file2` will delete both files.

1.1.8 Intermediate summary

All the `UNIX` commands related to filesystem navigation and file operations that we have seen so far[48] (Secs. 1.1.4-1.1.7) are conveniently summarized for you in Tab. 1.1. You can also find the corresponding material discussed/summarized in the lecture notes, LN@006 and LN@011-012.

Recall that for all the listed commands the files/directories can be always specified either by absolute or by relative paths. For example, if your current directory is ∼`/ex1`, `mv kiwi.txt ..` (relative paths), `mv /home/user/ex1/kiwi.txt /home/user` (absolute paths) and `mv kiwi.txt /home/user` (mixed) are all valid and will produce the same result.

As a last note, you will (often) observe that when you ask for something that is not possible, `UNIX` informs you with a warning message that the operation could not be carried out and why. Some examples (there are many others) are trying to `cd` into a file, to `mv` or `cp` a directory to a

[43] There is no "recycle bin" when you type a `UNIX` command that erases something. Once a file is gone, it is gone for good. In serious academic/industrial research groups, the system runs every night a daily backup, that saves somewhere a copy of the contents of the directories of all users. But no such backup is made for the ETHZ-student accounts. Note, however, that the file manager may have a recycle bin (not sure; and it will only work if you decided - against our advice - to keep using that obsolete tool).

[44] The command `ls` should print the contents of ∼`/ex1`, namely `my_dir`, `ananas.txt` and `mango.txt`. The command `ls my_dir` should print the contents of ∼`/ex1/my_dir`, namely `copy_of_my_dir`, `ananas.txt` and `banana.txt`.

[45] In the file-manager window, you can delete files by clicking on them and pressing the `<DEL>` key, or by right-clicking and selecting "delete".

[46] Was looking for synonyms of "stubbornly" and "angry" - to vary a bit the adjectives.

[47] Like in `cp -r`, the option `-r` stands for "recursive".

[48] Additional options for the command `ls` have also been included, which will be discussed in Sec. 1.1.9.1, as well as the command `chmod`, which will be discussed in Sec. 1.1.11.3.

file, or to `cp` or `rm` directories without the `-r` option. The same will generally[49] happen when you mistype a command, *e.g.* type `la` instead of `ls`.

At this point, you should be able to fly `UNIX` with your own wings. So, please: take a deep breath, gather your strengths, and... click off to hell the file manager. Real hackers use `UNIX`, and leave the file manager to the dummies...

1.1.9 UNIX commands

1.1.9.1 Command syntax

All `UNIX` commands have the same syntax (see LN@002):
`command [-options] [object1] [object2] ...`
Here, what is in square brakets is optional (may be needed/added or not). The one or more objects that are specified are referred to as the *arguments* of the command. The *options* serve to modulate the action of the basic command[50]. We have already seen one of them, the option `-r` to `cp` and `rm`, by which we allow the command to operate on directories and not only on files. In Tab. 1.1, we have also listed possible options of the command `ls`, namely `-R` (list recursively, *i.e.* list the entire contents of the directory always going into all subdirectories), `-p` (add a trailing `/` at the end of directory names), `-l` (list contents with more details), `-a` (also list hidden files/directories), `-t` (sort the list by last-modification date/time, from most recent to most ancient), and `-r` (reverse list order). These options are useful, but are not essential for your survival. If you are interested, more information can be found check for more info on the net or using `man ls`. If you want to use multiple options for a command, you can either list them one after the other, or combine them in one string[51]. For example, to print the contents of the current directory with details and in reverse order of last-modification date/time (*i.e.* from most ancient to most recent), you can type `ls -l -t -r`, or you can simply type `ls -ltr`.

1.1.9.2 Manual entries

There are many `UNIX` commands, and each of them usually has many options. It is a good idea to keep in mind the most common commands/options, such as those listed in the different tables of this document (at least, these are the ones you are expected to master for these exercises and for the final exam). But there is no way to keep in mind all commands/options. For this reason, `UNIX` offers an extremely convenient system of *text-based manual*, which you can access by the command `man`. For example, `man ls` will print to the screen the information on the command `ls`, its function, its syntax and all its options. You can conveniently browse through this page using the keys arrow down `<↓>` (one line down), arrow up `<↑>` (one line up), `<SPACE>` (one page down) and `<u>` (one page up). And when you are done, type `<q>` and the page will disappear. When you have doubts on a command or an option, this should be a reflex: look at the manual page! As a little exercise, imagine you don't remember what the option `-r` does for `rm`, and try to find out using the on-line manual.

[49] Since there are many `UNIX` commands, mistyping a command can also result in calling the wrong command. Often, you will still get an error message as the wrong command does not understand the arguments you have given to it.

[50] For some commands, the options you select may influence the number of arguments you have to give.

[51] This is generally valid for the most basic commands, but not an absolute rule. Some commands expect a specific ordering of the options and associated arguments.

csh	invoke the c-shell variant of `UNIX` (to be done only once at the start when you open a new terminal window)
pwd	print the absolute path of the current (working) directory
cd dir	change the current directory to dir (dir must be a directory)
cd ∼ *or just* **cd**	change to your home directory
cd ..	change to the parent directory (one level up from the current directory)
cd .	change to the current directory (*i.e.* no change)
ls	list the contents of the current directory (files and subdirectories)
ls dir	list the contents of the directory dir
ls [file/dir . . .]	more general, with multiple file/directory names (they will be listed in sequence; for a file, the relative path is simply printed if the file exists)
ls -R	list recursively the entire contents (including subdirectories, sub-subdirectories, *etc...*)
ls -p	list contents adding a trailing `/` at the end of directory names
ls -l	list contents with more details (long format including permissions, owner/group, sizes and last-modification date/time)
ls -a	list contents including hidden files/directories (*i.e.* those with a name starting with a dot)
ls -t	list contents sorted by last-modification date/time (most recent to most ancient)
ls -r	list in reverse order (*e.g.* useful in combination with `-t`)
mkdir [dir . . .]	create new (empty) directory (several directory names may be specified)
touch [file . . .]	for a file that does not exist, create an empty file with the given name, otherwise, update last-modification date/time of the existing file (several filenames may be specified)
cp [source . . .] dest	copy source to dest (source must be an existing file [not a directory], dest can be a new filename or an existing directory; several source files may be specified)
cp -r [source . . .] dest	copy source to dest (source may now be a directory, dest can be a new file/directory name or an existing directory; several source files/directories may be specified)
mv [source . . .] dest	move source to dest (source must be an existing file/directory, dest can be a new file/directory name or an existing directory; several source files/directories may be specified)
rm [target . . .]	delete target (target must be an existing file [not a directory], several target files may be specified)
rmdir [target . . .]	delete target (target must be an empty directory, several target directories may be specified)
rm -r [target . . .]	delete target recursively (target must be an existing file/directory, non-empty directories are deleted with their entire content; several target files/directories may be specified)
chmod perm file/dir	change the permissions of the file/directory according to perm

Table 1.1: Basic `UNIX` commands for filesystem navigation and file operations. The notation "[object . . .]" in this table means that one or more objects may be specified (the brackets themselves should not be typed! - just list none, one, or multiple objects after the command, separated by spaces). For all the commands, the files/directories can be always specified either by absolute or by relative paths.

1.1.9.3 Standard channels

All UNIX commands have access to three *standard channels* (see LN@013): the *standard input* (where they can read data), the *standard output* (where they can write data), and the *standard error* (where they can write warning/error messages). If you don't do anything special, the standard input is the keyboard (what you type after you have given the command and hit <ENTER>), and the standard output and error are the terminal (what is printed in the terminal window after you have given the command and hit <ENTER>). All the commands we have seen till now (*e.g.* ls) do nothing with their standard input (*i.e.* they don't expect you to type anything after giving the command). Some of the commands we have seen till now (*e.g.* ls) send something to their standard output (*i.e.* they print something to the terminal, *e.g.* for ls, the directory contents), and some send nothing to it (*e.g.* cd). Finally, all these commands send something to the standard error if they report a warning for some operation they cannot carry out.

One of the powers of UNIX is that the above defaults (input from keyboard, output/error to screen) can be changed. We can replace them by files and even link them to other commands. This is called input/output/error *redirection*.

The redirection of the standard output to a file is done using the symbol >. Just try the following in the directory ∼/ex1

```
[user@comp ∼/ex1]$ ls
my_dir ananas.txt mango.txt
[user@comp ∼/ex1]$ ls my_dir
ananas.txt banana.txt
[user@comp ∼/ex1]$ ls my_dir > ls.out
[user@comp ∼/ex1]$ ls
my_dir ls.out ananas.txt mango.txt
[user@comp ∼/ex1]$ cat ls.out
ananas.txt
banana.txt
```

Let's look step by step. The ls shows us (on screen) what we have in the current directory. The ls my_dir shows us (on screen) what we have in the subdirectory my_dir. The ls my_dir > ls .out does the same, but its standard output (what would be printed to the screen) is redirected to a file ls.out instead. The next ls shows us that indeed, this file was created. Finally, we have introduced a new command cat (shorthand for "concatenate"). This command prints the contents of the file given as argument, *i.e.* in this case, the contents of the newly created ls.out. Note that redirecting the output of a command to a file that already exists will cause the existing file to be overwritten.

The redirection of the standard output to a file can also be performed in an appending mode (*i.e.* adding at the end of an existing file instead of overwriting it), which is done using >> instead of >. For instance, let's try the following

```
[user@comp ∼/ex1]$ pwd >> ls.out
[user@comp ∼/ex1]$ cat ls.out
ananas.txt
banana.txt
/home/user/ex1
```

The standard output of pwd has been redirected and appended to the file ls.out, resulting in the expected output. Note that appending the standard output of a command to a file that does not exists will cause the file to be created (in this special case, > and >> produce exactly the same result).

In the above, we used the command cat with one argument, the file to be printed. When you give no argument to the command cat, it will use its standard input instead of a file. This is the

first command we encounter that uses a standard input. And we can use this feature to create a text file from something we type on the keyboard. Let's try it out with the example below. Note that the lines starting with % indicate that you have to type the text yourself on the keyboard[52] (but don't type the % themselves!)

```
[user@comp ~/ex1]$ cat > poem_1.txt
%les sanglots longs
%des violons de l'automne
%<CTRL-d>
[user@comp ~/ex1]$ cat poem_1.txt
les sanglots longs
des violons de l'automne
```

There is another way to make a text file, which is the command `echo` (shorthand for... well... just for "echo"). This command takes a text you give to it as argument, and sends it to its standard output. Let's try it out

```
[user@comp ~/ex1]$ echo "blessent mon coeur" > poem_2.txt
[user@comp ~/ex1]$ echo "d'une langueur monotonne" >> poem_2.txt
[user@comp ~/ex1]$ cat poem_2.txt
blessent mon coeur
d'une langueur monotonne
```

Note that the text given as argument to `echo` must be surrounded here by double quotes (the reason why you have to do that will be explained in Sec. 1.1.11.2).

Above, we saw what `cat` does with zero or one argument. You can also give `cat` more than one filename. In this case, it will print (concatenate) the contents of all files in sequence to its standard output. The commands below will allow you to reconstruct the four lines of the poem in two different ways

```
[user@comp ~/ex1]$ cat poem_1.txt poem_2.txt > poem.txt
[user@comp ~/ex1]$ cp poem_1.txt poem_bis.txt
[user@comp ~/ex1]$ cat poem_2.txt >> poem_bis.txt
```

Feel free to verify with `cat` that the two versions `poem.txt` and `poem_bis.txt` are indeed identical. And beyond `UNIX`, we hope that you also realize the *historical* significance of seeing these these four verses reunited[53].

Now comes one of the most powerful features of redirection: you can connect the standard output of a command directly to the standard input of another command. This is done using the symbol `|`, which is called a *pipe*. To give an example, the command `grep` (sounding like[54] "grab") will search its standard input for all lines containing a text given as argument, and print these lines to its standard output. Now imagine we look for the lines containing the letter "s" in our poem. We could simply do it with

```
[user@comp ~/ex1]$ cat poem.txt | grep "s" > poem_with_s.out
[user@comp ~/ex1]$ cat poem_with_s.out
les sanglots longs
des violons de l'automne
blessent mon coeur
```

[52] Type `<ENTER>` to go to the next line of text. And type `<CTRL-d>` on the last empty line (press simultaneously the `<CTRL>` key and the `<d>` key) after the last `<ENTER>` to tell `UNIX` that you are finished with the keyboard input.

[53] If you don't, just type "Verlaine and Overlord" in your favorite web search engine...

[54] In fact, `grep` stands for "global regular expression print".

Here, `cat` sends the contents of `poem.txt` to its standard output. But this output is redirected (pipe) to become the standard input of `grep`. This second command searches its standard input for lines containing "s" and sends these lines to its standard output. This standard output is then redirected to the file `poem_with_s.out`, which has the expected content. If we now wanted to know which of these lines also contain the word "mon", we could use `cat poem_with_s.out | grep "mon"`. But we could also do it in one go with multiple pipes

```
[user@comp ~/ex1]$ cat poem.txt | grep "s" | grep "mon"
blessent mon coeur
```

There are many `UNIX` commands like `grep` that can perform a processing of their standard input into a standard output (modulated by their options and arguments). They are called *filters* and we will see more of them in Sec. 1.1.12. By assembling filters using pipes, an experienced `UNIX` user can build extremely powerful *pipelines* for processing files. Note that `grep`, like most filters, also accepts a syntax where the input is a filename given as argument. In this case, it processes the file instead of its standard input (see Sec. 1.1.12.11 for more details about this). For example, `grep "s" poem.txt` is equivalent to `cat poem.txt | grep "s"`.

For completeness, one should mention a few additional (less important) features of redirection. First one can redirect the standard input of a command (so that it comes from a file and not from the keyboard) using the symbol `<`. For example, you could write `grep "s" < poem.txt`. But since this is exactly equivalent to `cat poem.txt | grep "s"`, one can forget about this additional notation. Second, one can redirect the standard error of a command together with its standard output to a file (instead of the screen), using `>&`. There also exists `>>&` to redirect the standard output and error in appending mode. When you use `cmd > file`, possible error messages will still go to the screen. If you use `cmd >& file`, possible error messages will go to the file, along with the normal output. Third, there exists a cool command `tee` (shorthand for "T", as it is a three-point connector) that copies its standard input to its standard output (just like `cat` with no arguments), but also sends a copy of it to a file given as argument. For example, `cat poem.txt | grep "s" | tee poem_with_s.out | grep "mon"` would do the same as in the last example above, but also create the file `poem_with_s.out` of the previous example.

The main information about channel redirection is summarized in Tab. 1.2.

1.1.9.4 Command path, aliases, shell scripts

Most of the `UNIX` commands are actually files of one special type called *executables*. Often, an executable is a *binary* file[55], *i.e.* not human readable[56] (in opposition to a *plain text* file, which you can read). An executable file contains a program that the computer can execute. This program will receive the options and arguments you give along with the command.

To see where the executable file corresponding to a given command is located, you can use the command `which`. For example

```
[user@comp ~/ex1]$ which cat
/usr/bin/cat
```

Many of the basic `UNIX` commands are located in `/bin` or in `/usr/bin`. You can try `ls /bin /usr/bin` to get a feeling how many there are. How does `UNIX` know where the executable file of a command you just typed is located? Well, it searches a standard set of directories, specified in a

[55] There are executable files that are not binaries. An example is `UNIX` shell scripts (nowadays, `python` scripts are also very popular). A *script* is a human-readable program, that is `interpreted` (*i.e.* translated to assembly during the execution) and not `compiled` (*i.e.* translated to assembly before the execution).

[56] If you want to see what it looks like, type for example `less /usr/bin/ls`, which will show you the contents of the executable file for the `UNIX` command `ls`. Use `<SPACE>` to go down one page, and `<q>` to exit when you are sick of it.

echo text	print the given text to standard output (the text is generally surrounded by quotes)
cat [file ...]	if no argument is given, copy standard input to standard output; if one file is given, print the file contents to standard output; if several files are given, print the file contents in sequence (concatenate) to standard output
tee file	copy the standard input to standard output as well as to the indicated file
> file	redirect standard output to the file (overwriting the file if it already exists)
< file	redirect standard input from the file (the file must exist)
>> file	append standard ouptut at the rear of the file (creating the file if it does not exist)
>& file	redirect standard output and error to the file (overwriting the file if it already exists)
>>& file	append standard output and error at the rear of the file (creating the file if it does not exist)
| cmd	redirect standard output to the standard input of a following command cmd (pipe)

Table 1.2: Commands and symbols relevant for the use of standard channels. The notation "[object ...]" in this table means that one or more objects may be specified (the brackets themselves should not be typed! - just list none, one, or multiple objects after the command, separated by spaces). Note that `grep` has been omitted, as it is listed later in Tab. 1.3.

variable called the *path variable*. You can see its contents by typing `echo $path`. Listed are all the directories where `UNIX` searches for the executable associated with a command you have just typed.

When an executable is in a directory that is not listed in the path variable, it will not be found when you simply type the command. But you can still execute it by giving its absolute path. Here is a simple example

```
[user@comp ~/ex1]$ cp /usr/bin/cat fox
[user@comp ~/ex1]$ fox poem_1.txt
fox: Command not found.
[user@comp ~/ex1]$ ./fox poem_1.txt
les sanglots longs
des violons de l'automne
```

We have copied the executable file corresponding to the command `cat` to your home directory, under the name `fox`. But when you try to execute `fox`, `UNIX` does not find it because it is not in the path where it looks for standard commands. By giving the full path of the command as `./fox`, where we recall that `.` means[57] "in the current directory", we can now execute a `fox`, which obviously does the same as executing a `cat`[58].

Some other commands are built directly in the `UNIX` operating system (more precisely, in the c-shell that you employ now), so, the corresponding program is not a file but really a part of the shell program. You can try for instance

```
[user@comp ~/ex1]$ which which
which: shell built-in command.
```

[57] This is a second example where the symbol `.` is useful (in Sec. 1.1.6, we promised you we would give two; mission accomplished!).

[58] In particular, both types of executions are likely to draw the attention of the Tierschutzbund.

Finally, some commands are so-called *aliases*. This means that the command has been bound to a specific text. You can try for instance

```
[user@comp ~/ex1]$ which ls
ls: aliased to ls --color=auto
```

This means that when you type `ls`, the shell translates it immediately to `ls --color=auto`. The added option makes sure that when you type `ls`, you automatically (*i.e.* without putting the option yourself) get a nice color coding when the directory contents is printed. You can see all the aliases the c-shell is using for you at the moment by simply typing `alias`.

This is about all you need to know about command path and aliases. But at some later point, you may wonder: (*i*) how to change the path variable (for example, to add a directory where you will write your own commands, *i.e.* your `UNIX` scripts [see below] or `C++` programs); (*ii*) how you can delete/change/add specific aliases (for example, if you wish that `ls` has the default option `-l` to always print detailed lists). Then you might consider looking in the Appendix (Sec. 1.A.1).

At some (other) later point, you may also encounter the problem that you perform the same operation (series of commands) many times, and it is a lot of typing. Then you will be ready to consider writing *shell scripts*. A shell script is a series of `UNIX` commands you could type one after the other in the terminal window, but that you put in a file instead. In addition, the file must start with the line[59] `#!/bin/csh`. When you execute the shell script, it automatically executes all the commands listed in the file one after the other. A simple example, try (the lines starting with `%` indicate that you have to type the text yourself on the keyboard; but don't type the `%` themselves)

```
[user@comp ~/ex1]$ cat > bla
%#!/bin/csh
%echo "I am desperately tired"
%<CTRL-d>
```

Now, everytime you enter the command `./bla` it will print the above desperation message (don't run it too often, it would ruin your good mood). Writing shell scripts is a smart way to do file operations/processing extremely efficiently (there are also mechanisms to use command-line arguments and options, variables, conditions and loops). If you are interested, check for more info on the net!

1.1.10 Intermezzo: getting provided exercise material

At this point, you know enough of `UNIX` to perform an operation that you will have to repeat at the start of each new exercise: copying to your home directory files provided for a given exercise on the Informatik I web-site, `http://www.csms.ethz.ch/education/InfoI`. For reference, the procedure is documented in Ex. 0, the *starting document for students*, Sec. 0.5. For the next exercises, you can refer to this text. But for this first exercise, the procedure is a bit different, and we describe it here. Please, *carefully* carry out all the steps below:

1. Type `cd` to make sure you are in your `UNIX` home directory.
2. Type [60]

[59] This line says that the script is written for the c-shell variant. It is also a good habit (but not compulsory) to end your scripts with the command `exit`. Finally, you have to give them execute permission before being able to run them, see later Sec. 1.1.11.3.

[60] Sorry, this is a long line (actually, everything before `doc` is the choice of ETHZ Informatikdienste; only the rest you can blame on us). Anticipating on Sec. 1.1.15: if you make a mistake, you can recall the line by hitting the `<↑>` key multiple times (until the line shows up again), then correct what was wrong, then hit `<ENTER>`.

```
curl https://ethz.ch/content/dam/ethz/special-interest/chab/physical-chemistry
/csms-dam/doc/exe/ex1.tar > ex1.tar
```

3. Type `tar -xpvf ex1.tar`. You should see a list of files. If you see instead an error message like "this is not a tar archive", it means you mistyped the address; then try again at the previous step (but, to save typing, look at the last footnote before!).

4. Finally type `ex1P/setup/setup`

And if you are curious to know what you just did, read this footnote[61].

1.1.11 File properties and manipulation

1.1.11.1 Filename extensions

Go to the directory ~`/ex1P/messy` and check what is in there (using `ls`). There are files with different *filename extensions*, like `.dat`, `.log`, `.pdf` and `.txt`. Filename extensions are commonly used as a trick to indicate at the level of the filename what the file may contain. In fact, this is not limited to `UNIX`. Your Windows or MacOS PC also uses filename extensions. But generally, by default, these are not shown in the names under the different icons on the desktop[62]. Instead, the extension is used to choose the type of icon that will be displayed[63] (*e.g.* a pdf-document icon if the file ends in `.pdf`). Although it is common (and very useful) to use filename extensions, it is not compulsory. Note also that `UNIX` executables (`UNIX` programs, common applications and user programs) normally have no extension[64].

1.1.11.2 Wildcards

As the name indicates, the directory ~`/ex1P/messy` is a bit of a mess, mixing all sorts of files. Let's say we want to clean up. First, we would like to make a new directory `PDF` where we will gather all the `.pdf` files. We could make the directory (`mkdir PDF`) and then move all the files (`mv 1.pdf 2.pdf 3.pdf 4.pdf 5.pdf PDF`). This is a lot to type. Fortunately, `UNIX` recognizes so-called *wildcards* (like a "joker" in the scrabble game, if you want), *i.e.* notations that are automatically expanded to cover all the files matching a certain pattern (see LN@004). The most common wildcard is `*` which matches "any sequence of characters" (including no character at all as a special case). So, the job is done in one go with

```
[user@comp messy]$ mkdir PDF
[user@comp messy]$ mv *.pdf PDF
```

This works because `*.pdf` is expanded to a list of all files in the directory with a name ending in `.pdf`.

[61] The `curl` command is a `UNIX`-based way to read a file from the web (instead of using a web-browser). The page you access contains a file `ex1.tar`, which is an archive. An *archive file* is a single file that packages together a collection of files, possibly along with information on its original directory structure or/and a compression of the data. The command `tar` will extract the archive into a new directory `ex1P`. Finally, the shell script `ex1P/setup/setup` will perform a few set-up and clean-up operations. In particular, it will make sure that you never again have to type this absurdly long web address.

[62] If you like (I personally do), you can change this default in the system settings of your computer, so that the extensions are also shown in the filenames under the icons.

[63] It also often determines the default application (program) that will be used to open the file when you double-click on it. Note that this concept of "default application" does not exist in `UNIX` (but the GUI includes this feature, so if you still use icons and double-clicking, some default application will be invoked).

[64] On a PC, they usually have a `.exe` extension.

Another wildcard is `?`, which matches "any single character" (this time, *not* including no character at all, *i.e.* there must be *exactly* one). So, we can move all the `.log` files concerning a person with a short (three characters) name into a new directory `LOG_short` as

```
[user@comp messy]$ mkdir LOG_short
[user@comp messy]$ mv ???.log LOG_short
```

You can also restrict the list of allowed characters using lists or ranges between square brackets. For example, `[ABC]` matches `A`, `B` or `C`, `[0-9]` matches any digit, and `[a-cx-z]` marches `a`, `b`, `c`, `x`, `y` or `z`.

Note that the use of wildcards is general, *i.e.* by no means restricted to `mv`. For example, if you want to list `.txt` files only, type `ls *.txt`. And if you want to find the number of lines containing "hello" in the files starting in `t` and ending in `.txt`, you can use

```
[user@comp messy]$ grep "hello" t*.txt
tac.txt.hello world
tic.txt:hello me
toc.txt:hello you
```

The characters that are interpreted in a special way (wildcards, but also other purposes) by `UNIX` are

```
$ [ ]   ! | ‘ ’ " * &  ? ( ) < >
```

Note that these characters are not allowed in file/directory names. The interpretation of these characters is very useful, but it can sometimes be annoying, because it means that certain characters will be interpreted in a certain way even when you want to use them literally. For example, for writing a star `*` into a file `my_star`, you might think of using `echo * > my_star`. Please, try and enjoy the result (with `cat my_star`). You can handle this by preceding these characters with a[65] `\`. So, for example, `echo \* > my_star` will work perfectly. You can also handle it by surrounding these characters (or the text containing them) with single or double quotes. So, for example, `echo "*" > my_star` or `echo '*' > my_star` will also work perfectly[66].

Using quotes is also useful when you need to give an argument that is a text containing white spaces. For example, `cat file | grep some thing` will not work, because `grep` will believe that you are giving two arguments (the first being seen as the text, but the second being interpreted a filename). But `cat file | grep "some thing"` will work just fine. And `cat file | grep " some thing "` (extra space at the left and right) will also make sure that space-separated words are found (*e.g.* the text "awesome thingy" will no longer match).

Feel free to play further with wildcards if you want to do more clean-up in the directory `~/ex1P/messy`. When you are tired of it, just move to the next section.

1.1.11.3 File/directory permissions

On your own PC (at home or laptop), you are probably the only user and you want to be able to access everything[67]. In contrast, `UNIX` systems are often shared by many users[68]. Formally,

[65] In the jargon, one calls this *escaping* a wildcard. And you can even escape the escape (a bit like a prisonner who would refuse to take part in an evasion project), *i.e.* if you need to print a backslash, use `\\` (which will prevent it from being interpreted as a wildcard escape).

[66] If the text includes a single quote, then use double quotes to surround it. If the text includes a double quote, then use single quotes to surround it. If it includes both, then just append the two notations. For example, to print `"trop d'amis"` you can use `echo '"'trop d"'"amis'"'`.

[67] Well, if your PC or Mac is nicely set up, you probably have two accounts: one for you as a user (who cannot mess up too many things) and one for you as an administrator (who could mess up everything).

[68] For example, the computer you now work on can mount (over the network) the home directories of *all* ETH users.

in UNIX, you often share a unique filesystem (everything in the root directory and down) with literally thousands of people. Clearly, access to specific files/directories has to be regulated, which is done by setting *permissions*. The topic of file/directory permissions is discussed in the lecture notes, see LN@008-LN@010 and LN@022.

To explore a bit the issue of permissions, we will go to the directory ~/ex1P/perm. Go there, and do a pwd.

```
[user@comp ~/ex1]$ cd ~/ex1P/perm
[user@comp perm]$ pwd
/tmp/user/ex1P/perm
```

Something should definitely disturb you... Do you see what? The current directory should be /home/user/ex1P/perm and not /tmp/user/ex1P/perm. The reason is that you went through a *link*, the UNIX equivalent of a portal (or of a wormhole, if you prefer). The file /home/user/ex1P/perm does not exist, it is a pointer to another file location /tmp/user/ex1P/perm. If you are curious why we did this, it will be explained in a footnote at the end of this section[69]. For the moment, don't panic and just recover from the dizziness of the warp: it does not affect any of the things below.

Check what is in the current directory using ls with the option -l, which gives you more details on the files/directories (long format including permissions, owner/group, sizes and last-modification date/time; note that we simplified the output below to only keep the three essential pieces of the listing)

```
[user@comp perm]$ ls -l
-rwxrwxrwx user domainusers all_for_all
-r-------- user domainusers read_only.txt
-rw------- user domainusers writeable.txt
-r-x------ user domainusers executable
```

In UNIX, every file/directory has an *owner*, who is allowed to regulate the permissions for this file/directory. The owner belongs to a *group*, and everyone else is referred to as *the others*. In the listing above, the second entry is the owner, which should be you, and the third entry is the group you belong to (yes, whether you like it or not, you belong to the group domainusers). The first entry lists in sequence: object type (one character, - for a plain file or d for a directory[70]), then the permissions for the user (*i.e.* the owner), the group and the others (three characters each). The three characters are r (read permission), w (write permission) and x (execute permission).

From this listing, you can conclude that everyone has every permission on the file all_for_all. For the three other files, the group and the others have no permissions whatsoever. And you, as the user, have read permission on all files, plus write permission for writeable.txt and execute permission for executable. You can verify all this easily. To try to read a file, use cat file. To try to write into a file, use echo "hello" > file. To try to execute a file[71], use ./file. Note that if you give to the group or/and to others the permission to execute one of your files, they

[69] And if you are not curious, it will still be explained in a footnote at the end of this section. Just you won't read it.

[70] There are five other possible types of characters/objects, l for a symbolic link, p for a named pipe, s for a socket, b for a block device, and c for a character device, but we can leave these ones to IT people. Maybe we could make an exception for the symbolic link, which may be fairly useful at times. A link is an object in the filesystem that points to a directory somewhere else in the filesystem. Remember that you just crossed one to get into /tmp/user/ex1P/perm (for sure you remember: probably your head is still buzzing after the sudden warp). If you are curious, try ls -l ~/ex1P/perm: the l at the line start says it is a link, and the -> indicates where it points to.

[71] Recall that you need the initial ./ to tell UNIX that the command is in the current directory (as this directory is not included in the path variable where UNIX looks for standarrd commands).

can run the corresponding program, but it will still "act" as them and not as you in terms of permissions on your files and directories[72].

Since you are the owner of these files, you are allowed to change all the permissions as you wish, for yourself, for the group and for the others. The command to do this is `chmod` and it has two possible syntaxes. The first syntax involves three elements specifying who is concerned (`u` for user, `g` for group, `o` for other, `a` for all), if you add or remove a permission (`+` for add, `-` for remove), and what permission (`r`, `w` or `x`). For the first and third elements, you are allowed to use multiple letters. For example, try

```
[user@comp perm]$ chmod u-r read_only.txt
[user@comp perm]$ chmod a+r writeable.txt
[user@comp perm]$ chmod go+rx executable
[user@comp perm]$ ls -l
-rwxrwxrwx user domainusers all_for_all
---------- user domainusers read_only.txt
-rw-r--r-- user domainusers writeable.txt
-r-xr-xr-x user domainusers executable
```

Now everyone can read `writeable.txt`, but you are still the only one who can modify it. And everyone can read and execute `executable`. Finally, you can no longer read `read_only.txt`. But since you are still the owner, the file is not lost: you just have to change the permission back.

There is a second syntax one can use with the `chmod` command, which relies on a three-digit octal string to change all permissions in one go. The three digits refer in turn to user, group and other (remember: `ugo`). And for each digit, the octal code refers in turn to read, write and execute (remember: `rwx`). To get the octal code, add 4 if read, 2 if write, 1 if execute. For example, read only gives 4, read+write gives 4+2=6. read+write+execute gives 4+2+1=7. Now try

```
[user@comp perm]$ chmod 764 read_only.txt
[user@comp perm]$ ls -l read_only.txt
-rwxrw-r-- user domainusers read_only.txt
```

Permissions can also be set for directories. In this case, the meaning of the three types of premissions are: (*i*) read gives the right to read the names of files in the directory (but if alone, no additional information); (*ii*) write gives the right to modify entries in the directory (creating files, deleting files, renaming files); (*iii*) execute gives the right to access file contents and metainfo (but alone, not to list the directory). This is a bit paradoxical. With execute-only permission on a directory `dir` containing a file `file`, you can do `ls dir/file` but not `ls dir`. If you want to avoid weird problems, it is advisable to keep the three permissions open on all your directories.

For completeness, let's still mention two last commands. The command `chgrp` permits to change the group of a file/directory so that it belongs to another group (this only makes sense if you belong to two or more groups). And the command `chown` permits to change the owner of a file/directory so it belongs to another owner. For obvious reasons, only a *superuser* (administrator) can use `chown`. And only a superuser or the owner can use `chgrp`.

And as a last remark: it is not because you give execute permissions to a file that it becomes a meaningful executable, *i.e.* a program that the computer can understand. For example, just

[72]This means, for example, that the program will not be allowed to delete one of your files unless this file has write permission for them as group or others, which is a nice safety precaution. However, there exist mechanisms called `setuid` and `setgid` which allow a program of yours that is run by others to "act" as a you or as a member of your group, respectively. When this is enabled, you will see a `s` instead of the `x` in the corresponding permissions. We will not teach you how to do that. First you don't need it. Second, it is dangerous! For example, if you have file in your directory that has both setuid and write permission for the others, anyone can replace the file by their own program, and let it "act" as you in your account. This type of mechanisms, exploiting a flaw in a software that permits an anonymous user to gain special privileges in the system, is one of the common mechanisms of action of malwares, called *privilege escalation*.

above, you gave yourself execute permission for `read_only.txt`. So, try to execute it with `./read_only.txt` and see what happens. The error message tells you that what is written in the file is not understood at all as a meaningful program. In contrast, the file `executable` is a meaningful executable (as you see from typing `./executable`), in fact this is a (very short) `UNIX` shell script (briefly discussed previously in Sec. 1.1.9.4).

Now for the reason why we warped you through hyperspace, look (or don't look) here[73]. After this trip, time to get back home. By now, you should know how[74].

1.1.12 Useful `UNIX` commands

Below is a compendium of particularly useful `UNIX` commands. Their functions are summarized at the end, in Tab. 1.3. For now, let's play a bit with them. For this, go to the directory `~/ex1P/play`.

1.1.12.1 diff

The command `diff` serves to compare two text files. If there is no difference, it will give no output. If some lines differ, it will print a report telling what lines were added, changed or removed.

Examine the contents of the text files `tiger.txt`, `tiger_bis.txt` and `wolf.txt`. Now try `diff tiger.txt tiger_bis.txt` and then `diff tiger.txt wolf.txt`

```
[user@comp play]$ diff tiger.txt tiger_bis.txt
[user@comp play]$ diff tiger.txt wolf.txt
1c1
< The tiger is an animal
---
> The wolf is an animal
3c3
< Its scientific name is panthera tigris
---
> Its scientific name is canis lupus
```

In the first case, we see no differences: the files have an identical content. In the second case, we get a report of the differences.

See `man diff` for more details on the available options and for the format of the reported differences. Note in particular the option `-y` which makes differences easier to spot (just try!). The command will also work on non-text files (*e.g.* pdf, executables, ...). In this case, it will only report if the files differ at all, with no additional details on the differences.

1.1.12.2 paste

The command `paste` serves to assemble two text files column-wise[75]. This can be very useful in science, when two separate files contain data that you wish to correlate. For example, `miss.dat`

[73] As we explained briefly earlier, the home directories of ETHZ users are stored centrally by ETHZ Informatikdienste and exported over the network to computers in and out of ETH (including the one you are working on right now). Some of these computers don't run `UNIX`, but Windows or MacOS. And the latter operating systems don't use the sophisticated permission system of `UNIX`. They actually only know about the write permission of the user, that you can suppress by making the file "read only". Thus, for compatibility reasons, there are no permissions for group and others in the home directories of ETHZ users, and the user permissions can only be set to either `rwx` (writeable) or `r-x` (read only). You can easily check this by doing `ls -l` on files/directories in your home, and observing that you cannot change the permissions beyond `chmod u-w` and `chmod u+w`. In contrast, the directory `/tmp` is local to your computer and fully `UNIX`-based. For this reason, we did our explorations in a copy of `ex1P/perm` within this directory `/tmp` (that was connected `~/ex1P/perm` *via* a link).

[74] A good old `cd` should do the job, right?

[75] In contrast to `cat` that contatenates the files line-wise, one after the other.

lists the age of Miss America from 1999 to 2008 (two columns: year and age), and `murder.dat` lists the number of murders by steam, hot vapors and hot objects in the US over the same period (also two columns: year and number). Now just try

```
[user@comp play]$ paste miss.dat murder.dat
```

Enjoy the nice correlation but remember that correlation is not the same as causality.

See `man paste` for more details on the available options. Note in particular that you can paste more than two files, and that you can use the options `-d` to specify how you want to separate the columns in the output.

1.1.12.3 wc

The command `wc` (shorthand for[76] "word count") prints the number of lines, words and bytes in a text file (standard input or specified by a filename argument). Let's try it on three files (with three slightly different but equivalent syntaxes)

```
[user@comp play]$ wc tiger.txt
  4  19 103 tiger.txt
[user@comp play]$ wc < tiger_bis.txt
  4  19 103
[user@comp play]$ cat tiger.txt | wc
  4  19 103
```

See `man wc` for more details on the available options. Note in particular the options `-l`, `-w` and `-c`, which restrict the output to only one of the three quantities.

1.1.12.4 head

The command `head` is a filter that prints the first lines of a text file (standard input or specified by a filename argument). Usually, it is given with the option `-n` which tells the number of lines you want to print (if you don't specify, the default is 10). For example, you can print the first two lines of `tiger.txt` as

```
[user@comp play]$ head -n 2 tiger.txt
The tiger is an animal
That eats meat
```

See `man head` for more details on the available options. Usually, the syntax `head -2 file` is also recognized as a short-hand to `head -n 2 file`. The syntax `head -n -2 file` can also be used to print all but the last 2 lines of a file.

1.1.12.5 tail

The command `tail` is a filter that prints the last lines of a text file (standard input or specified by a filename argument). Usually, it is given with the option `-n` which tells the number of lines you want to print (if you don't specify, the default is 10). For example, you can print the second and third lines of `tiger.txt` as

```
[user@comp play]$ cat tiger.txt | head -n 3 | tail -n 2
That eats meat
Its scientific name is panthera tigris
```

[76] Agreed: maybe not the smartest acronym this time...

This is a nice example of pipeline (previously discussed in Sec. 1.1.9.3). The command `cat tiger.txt` sends the text of the file to standard output. This output is piped to the standard input of `head -n 3`, which sends the first three lines of what it received to its standard output. This output is again piped to the standard input of `tail -n 2`, which sends last two lines of what it received to its standard output, which is the terminal window. Since the last two lines of the first three lines of `tiger.txt` are indeed the second and third lines of this file, this is exactly what we wanted. Little question: would `cat tiger.txt | tail -n 2 | head -n 3` have worked as well? The answer is here[77]. See `man tail` for more details on the available options. Usually, the syntax `tail -2 file` is also recognized as a short-hand to `tail -n 2 file`. The syntax `tail -n +2 file` can also be used to print all but the first 2 lines of a file.

1.1.12.6 grep

We have already mentioned `grep` before (Secs. 1.1.9.3 and 1.1.11.2). It is a filter that will search a text file (its standard input or specified by a filename argument) for all lines containing a text given as argument, and print these lines to its standard output. For example, you can count the total number of lines containing the word "animal" in `tiger.txt` and `wolf.txt` with

```
[user@comp play]$ grep "animal" tiger.txt wolf.txt | wc -l
4
```

See `man grep` for more details on the available options. Note in particular the options `-i` (make the search case-insensitive), `-v` (find the lines that do *not* match the text), `-n` (also report the line numbers at which the text was found) and `-c` (also report the number of lines in which the text was found). There are many others...

Note also that `grep` recognizes some wildcards[78] (similar to the wildcards for `UNIX` files; the syntax is sometimes similar but not exactly identical), which allows it match *patterns* rather than simple text. But if you like wildcard mechanisms, then better use `egrep` which implements a standard form of them called *regular expressions*. Regular expressions used in commands like `egrep`, `sed` and `awk` (see below) represent an extremely powerful mechanism to do about anything you want with text files. They are far beyond the scope of what you have to know for this course, but if you are curious, check for more info on the net!

1.1.12.7 sort

The command `sort` is a filter that sorts a text file (standard input or specified by a filename argument), by default alphabetically. For example, you can you can try

```
[user@comp play]$ cat tiger.txt | sort
Its scientific name is panthera tigris
That eats meat
The tiger is an animal
This animal has four legs
```

See `man sort` for more details on the available options. Note in particular the options `-n` (sort numerically instead of alphabetically), `-r` (sort in reverse order), `-f` (make the sorting case-insensitive), and `-k` (sort according to a speficic column, instead of from the start of the line).

[77] No. Here, `tail` takes the last two lines of the file, and `head` the (at most) first three lines of this (which is here only two, because there are just two lines). So, the output is now `Its scientific name is panthera tigris` followed by `This animal has four legs`.

[78] You can prevent this with the option `-F`.

1.1.12.8 uniq

The command `uniq` is a filter that removes all adjacent repeats of a given line in a text file (standard input or specified by a filename argument), *i.e.* it prints the lines that are unique plus one copy only of the lines that are repeated. For example, you can you can try

```
[user@comp play]$ cat tiger.txt wolf.txt | sort | uniq
Its scientific name is canis lupus
Its scientific name is panthera tigris
That eats meat
The tiger is an animal
The wolf is an animal
This animal has four legs
```

See `man uniq` for more details on the available options. Note in particular the options `-u` (print only the lines that are *not* repeated) `-d` (print only the lines that *are* repeated, one copy only of each), `-i` (make the checking case-insensitive), and `-c` (print each line of the file preceded by a count of the number of adjacent repeats).

1.1.12.9 sed, awk

The commands `sed` (acronym for "stream editor") and `awk` (shorthand for "Aho, Weinberger & Kernighan", who developed it in 1977) are filters that can perform a wealth of modifications in a text file (standard input or specified by a filename argument) processed line after line. These commands can be used in conjunction with regular expressions, and are amazingly powerful and versatile. Here, we will only show two very simple illustrative examples.

Using `sed`, you can for example replace all occurrences of a text in a file by another text. Try for instance

```
[user@comp play]$ cat tiger.txt | sed 's/animal/amazing creature/g'
The tiger is an amazing creature
That eats meat
Its scientific name is panthera tigris
This amazing creature has four legs
```

Using `awk`, you can for example reorder and perform mathematical operations on the columns of a file. Try for instance

```
[user@comp play]$ paste miss.dat murder.dat | awk '{ print $1, $4, $2, $2/$4 }'
1999 7 24 3.42857
...
```

Here, we have pasted `miss.dat` (two columns: year and age of Miss America) with `murder.dat` (two columns: year and number of murders of the specific category), resulting in a four-column file. Then, with `awk`, we have swapped the order of columns 2 and 4, and added a fourth column with the ratio of columns 2 and 4 (*i.e.* age/number).

Learning more about these commands goes far beyond the scope of the present introductory exercise. But if you are curious, check for more info on the net! See also `man sed` and `man awk` for the available options. And finally note that there are entire books dedicated to these two commands!

1.1.12.10 more, less

The commands `more` and `less` do more or less[79] the same thing. They are so-called *pagers*, *i.e.* commands that can display a text file (standard input or specified by a filename argument) on a page-by-page basis. We recommend to use `less`, as it allows to page not only down, but also up. Have a look at the file `immortal.txt` using the command `cat`. As the text is too long for your window, you will only see the end of it. Now try instead to use `less immortal.txt` and you will see the text one page at a time. You can go down/up one line using the keyboard arrows. And you can go down/up one page using the keyboard `<SPACE>` and `<u>`. To exit the pager, simply type `<q>`. Pagers can be added at the end of pipelines when the output is very long. For example, to check that `sort` works well, try `cat immortal.txt | sort | less`. See `man more` and `man less` for more details on the available options and other useful keyboard keys within the pager.

1.1.12.11 Last comment on filters

Most filters (like `head`, `tail`, `grep`, `sort`, `uniq`, `sed` and `awk`) also accept a syntax where the input is a filename (or multiple filenames) given as argument (the same holds for `wc`, `more` and `less`). In this case, they process the file (or the concatenated files) instead of their standard input. For example, `grep pattern file` is equivalent to `cat file | grep pattern`. Feel free to choose the way you prefer. We normally prefer the second syntax, which is more in the logics of filters and pipelines. But the first syntax may seem more logical if you think of the filter as a standalone command to process files. Also, it permits to list multiple files (which will be concatenated before the action of the filter), and it lets the command know about the filenames (which may be reported when you add specific options to the command).

1.1.13 Process scheduling

Most computers (even PCs) nowadays have more than one processor[80]. But even with a few processors under the hood, there are many more tasks to accomplish at the same time than there are processors. These tasks may include a few commands you give to the computer simultaneously, but also the very large number of tasks necessary for the system to run (and that you do not control explicitly yourself). This is handled automatically by *process scheduling*, which assigns at each instant to each processor a chunk of all the tasks to be performed by the computer.

Up to now, we have given the computer one task at a time, by typing a command, hitting `<ENTER>`, waiting for the job to be done, and then getting back the prompt for a new command. One simple way to give a second task simultaneously is to open another terminal window, and type the new command while the first one is still running in the first window. But you can also do this within a single window. For this, you have to say that the first command is to be run in the *background*. This is specified by adding the symbol `&` at the end of the command line. And you can check your background jobs by typing `jobs`. Try for instance

```
[user@comp play]$ cd ~/ex1
[user@comp ~/ex1]$ sleep 60 &
[user@comp ~/ex1]$ sleep 120 &
[user@comp ~/ex1]$ jobs
```

The command `sleep` just tells the computer to wait so many seconds. What you have done here is to send two jobs to the background (*i.e.* which will run without you noticing it), one that waits

[79] Ha,ha,ha!

[80] In particular, the quad-core configuration with four processors is very common for PC's.

man command	display manual information (action, arguments, options, input, output) on the command
which command	print the absolute path of the file containing the command (or reports if it is a shell built-in command or an alias)
alias	print a list of all the commands that are aliases
diff file1 file2	print line-by-line differences between file1 and file2, and the lines where the differences occur
cat [file ...]	if no argument is given, copy standard input to standard output; if one file is given, print the file contents to standard output; if several files are given, print the file contents in sequence (concatenate) to standard output
paste file1 file2	assemble file1 and file2 column-wise
wc [file ...]	count the number of lines, words and characters in the files (if no file given, process standard input)
head -n num file	print the first num lines of the file (if no file given, process standard input)
tail -n num file	print the last num lines of the file (if no file given, process standard input)
grep pattern [file ...]	search and print the filename and the lines matching the pattern (if no file given, process standard input)
grep -i pattern [file ...]	make the search case-insensitive
grep -v pattern [file ...]	search for the lines that do not match the pattern
grep -n pattern [file ...]	also print the lines numbers matching the pattern
grep -c pattern [file ...]	also print the number of lines matching the pattern
sort [file ...]	sort alphabetically the contents of the file (if no file given, process standard input)
uniq [file ...]	remove all adjacent repeats of a given line in a text file (if no file given, process standard input)
sed script [file ...]	perform line-by-line modifications defined by the script (*e.g.* text replacements) in the text file (if no file given, process standard input)
awk script [file ...]	perform line-by-line modifications defined by the script (*e.g.* column changes) in the text file (if no file given, process standard input)
more file	print the contents of the file one page at a time (press `<q>` to quit; if no file given, process standard input)
less file	print the contents of the file one page at a time, more fancy version (press `<q>` to quit; if no file given, process standard input)

Table 1.3: Some particularly useful `UNIX` commands. The notation "[object ...]" in this table means that one or more objects may be specified (the brackets themselves should not be typed! - just list none, one, or multiple objects after the command, separated by spaces).

a minute and one that waits two minutes. When the jobs are finished, `UNIX` will notify you with a "done" message, and they will disappear from the job list.

There is much more to process scheduling in `UNIX`, but it is not essential for you now. If you are curious, check for more info on the net!

1.1.14 Using the network

The command to access another computer from your current machine within the terminal window is `ssh` (shorthand for "secure shell"). Have a look at the name of your machine[81]. Now ask your neighbor in the computer room for the name of his own machine, which we will write `neigh` in the following. Try to type `ssh user@neigh.ethz.ch`. You will be asked to confirm that this new computer is trustworthy by saying[82] "yes". Then, type your password (note that for safety reasons, the password is not printed on the screen when type it). Finally, type `csh`. There you are: your terminal window is now connected to the neighbor's computer *via* the network (you should now see the name `neigh` instead of `comp` in your prompt), and all commands you type will be executed on this other machine (but the data in your home directory is still exactly the same). Note that your neighbor can detect you. Just ask her/him to type the command `who` in the terminal window. The output should report your presence on her/his machine. Now, come back to your computer by typing twice[83] `exit`.

Note that you can use the same mechanism to log into another computer that is outside the computer room (you just have to know its network address and have the appropriate permissions), and to log from the outside into one of the machines of the computer room[84] (this may be useful if you want to work a bit on your exercises from home between two sessions with the assistant).

Some more information on using the network can be found in the Appendix (Sec. 1.A.2).

1.1.15 Time-saving tips

To conclude this part on learning `UNIX`, here are a few time-saving tips. You don't need to know/use them, but if you do, you will see that they may reduce a lot the amount of typing you have to do to get things done.

1. When you do simultaneous work in different directories, it is often a good idea to open multiple terminal windows and set different current directories for them. Then you swap between windows, rather than changing the current directory all the time in one window. Other advantage: you can use the usual cut/paste mechanism with the mouse to transfer text between the windows[85].

[81] The one we have written generically `comp` in the above text. The actual name is written (sticker) on the computer itself, and it is the same name you can read after the `@` in the command prompt.

[82] Your computer `comp` will receive a so-called RSA key from the neighbor computer `neigh`, which will be saved in your account. From this point, the `neigh` computer will be assumed "trustworthy" and you won't be asked again if you try to connect again at a later point.

[83] Once to exit the c-shell, and once to terminate the connection.

[84] From the outside, you cannot directly log into the machines of the computer rooms. First, you have to do `ssh user@slabN.ethz.ch`, where `N` can be 1, 2, 3 or 4. These are the four login machines of the ETHZ Informatikdienste (they are not in the computer rooms). Only from these machines can you further connect to any machine in the computer rooms, using `ssh user@slabhcibNNN.ethz.ch`, where `NNN` runs from `002` to `041`. Note that: (*i*) you don't need `vpn` to perform these operations; (*ii*) the first time you connect, you may have have to confirm that you trust the computer you try to access (no worries, you can trust ETH); (*iii*) the machines in our computer rooms are typically switched off (inaccessible) when no one is working on them; (*iv*) depending on what you have to do, it may be simpler/faster to directly work on one of the four login machines (which are always on).

[85] To do that, press the left mouse button, cover the text you want to copy from the first window, select the other window, and then do a right-click with the mouse.

2. You can give multiple commands in one single command line, by separating them with a `;`. For instance, `cd; ls` will bring you home and list the contents in one go.

3. You can see a list of all the recent commands you typed with the command `history`. And if you want to execute again a command without retyping it, you can use the usual cut/paste mechanism with the mouse to transfer from the history list to the current command line.

4. Even simpler, you can use the keys `<↑>` and `<↓>` to bring back previous commands to the command line directly. And when you have found the right one, just hit `<ENTER>` to re-execute the command. For example, to re-run the last command you gave, just hit `<↑>` and then `<ENTER>`. Quite convenient, isn't it?

5. You can also decide to use an old command as template and then slightly modify it. For this, use the `<←>` and `<→>` keys to move the cursor along the command line, change what you need to change, and hit `<ENTER>` when you are satisfied. Other useful key for this: `<CTRL-a>` (jump to line start), `<CTRL-e>` (jump to line end), `<CTRL-d>` or `<DEL>` (delete a character after the cursor), `<CTRL-k>` (delete all till end of line), `<CTRL-c>` (cancel and get a fresh empty prompt).

6. By using the `<TAB>` key on the keyboard the terminal tries to complete automatically what you are about to write. For example if you are in `~/ex1`, type `cd my` and then press `<TAB>`, the shell will automatically completed to `cd my_dir`, and you only have to hit `<ENTER>`. If there is more than one possibility how to complete your command, the `<TAB>` key will display all possibilities. For example if you just type `c` and press `<TAB>`, the shell will print all possible commands starting with a `c`. In this case, you should keep adding characters until the completion is unique.

8. In all cases, to quickly access information on commands and practical solutions to specific problems, keep in mind: (*i*) the command `man`; (*ii*) the use of internet (search engine with appropriate keywords).

7. And for advanced users: (*i*) define your own aliases (*e.g.* if you often use `history`, why not alias it to a one-letter command `h`); (*ii*) write your own `UNIX` shell scripts. For this, see Sec. 1.1.9.4 and Appendix (Secs. 1.A.1.

1.1.16 Useful applications

Congratulations: at this point, you know the essential of `UNIX`! Well, maybe you still need a bit of practice, which you will get next week in Sec. 1.2... And for the rest of this sandbox will do something that won't seem as exotic to you. Here, we will introduce you briefly to standard applications (programs) that you will need to use for the Informatik I exercises.

1.1.16.1 Web browser and e-mail

As a *web browser*, we recommend you to use *firefox* (see here[86] how to start it). Even if you use another web browser at home, you should be able to find your way easily with this one.

You can also use it to access your *e-mail*. The standard e-mail program for ETHZ staff and students is *outlook*. To access it, type the web link `www.mail.ethz.ch` in the browser, and go to "outlook web access". Log in with your `nethz` user name and password (the same you used to log into the computer). Now you will see your inbox (the mails you received). For the Informatik

[86] Click on "activities" in the top left corner of the screen. If it is not yet in your "favorite applications", select "show applications" (or press the `<WINDOWS>` key on the keyboard), and type "firefox" in the search bar.

I exercises, you will have to send your programs/graphs/reports as *e-mail attachments* to your assistant. The e-mail address depends on your exercise group (A-E for the different times of the week, but written in *lower case*, then 1 for HCP G24 or 2 for HIT F21, then `@igc.phys.chem.ethz.ch`;). If you have doubts about your group, ask your assistant or refer to the table "exercise groups" on the page `www.csms.ethz.ch/education/InfoI/InfoI_exercises`. So, for example, if your group is `B2`, the address will be `b2@igc.phys.chem.ethz.ch` (with the "b" in lower case). Let's try to send a test mail to your assistant, with the file `~/ex1P/hello/hello_my_assistant.txt` as attachment. Click on "new" and fill in the fields. Then click on the "paper clip" followed by "browse". Find the file `~/ex1P/hello/hello_my_assistant.txt`, click on it, and then on "open". You will see the path of the file left of the "browse" button. Click on "attach" and then, to send the e-mail, click on "send". Do not forget to "log off" from the outlook application when you are finished with writing e-mails. Note that you can of course send more than one file as attachment if you wish.

1.1.16.2 Text editing

We have already seen a number of ways of creating text files (for an empty file, `touch` in Sec. 1.1.7; for one or a few lines of text, `cat` with keyboard input or `echo` in Sec. 1.1.9.3). Clearly, this is not a good way for writing longer files, and also won't allow you to edit (modify) files that you have already written. For this, we need a text editor.

As a simple *text editor*, we recommend you to use[87] *gedit*. You can start `gedit` by just typing the name in the terminal window.

```
[user@comp ~/ex1]$ gedit &
```

The `&` has been added here to make sure `gedit` will run in the background. This will allow you to go on working in the terminal window instead of having it "blocked" by the process[88] `gedit`.

To open a file in the editor you can use the "open" button in the top left corner of the window, and then browse for the file you want to open. Another possibility is to start the text editor already with the command `gedit file &`, which will directly load the indicated file. When you are done modifying a text, don't forget to save the file, either using the "save" button on the top left (see also other options next to this button), or by pressing `<CTRL-s>`.

1.1.16.3 Graphic visualization

Scientists love graphs, and you will produce a number of them in the Informatik I exercises. To produce them based on data files, you need an application (program). As a simple *graphic visualization program*, we recommend you to use[89] `xmgrace`.

Let's take back the example of Sec. 1.1.12.2, and try to directly correlate in a graph the age of Miss America and the number of murders by steam, hot vapors and hot objects in the US over the same period. For this, we need to combine the data into a two-column file with the selected properties to correlate. We can do this with[90]

[87] Alternatives are `vi`, `kate` and `emacs`. But if you use one of these, we cannot guarantee that the assistant will be able to help you (everyone has a favorite). The most powerful one is probably `emacs` which is an absolutely amazing tool!

[88] If you once forget the `&` and your terminal is "blocked", just close the `gedit` window and try again (you can also press `CTRL-z` to bring the `gedit` to sleep, and then type the command `bg` to resume it in the background).

[89] Common alternatives are `gnuplot` or `R` (and many `python`-based tools). There are also many (more or less expensive) proprietary applications like `origin`, `mathematica`, `matlab` or `excel`. But if you use anything else than `xmgrace`, we cannot guarantee that the assistant will be able to help you (everyone has a favorite).

[90] If you don't understand what is done here, go back to Sec. 1.1.12.9.

```
[user@comp ~/ex1]$ cd ~/ex1P/play
[user@comp play]$ paste miss.dat murder.dat | awk '{ print $2,$4 }' > m_vs_m.dat
```

Now type `xmgrace &` in the terminal window, and you will see the window of `xmgrace` appear on the screen[91]. Open the menu "data", select "import", and then "ASCII". On the left you will see your directories. Find `~/ex1P/play`, select the file `m_vs_m.dat` on the right, then press "OK", and finally "cancel" to dismiss the window. Depending on the raw data, you might have to click on the "AS" (auto-scale) button in the left panel to see the whole data range. If you don't like to have your curve auto-scaled, you can also zoom in and zoom out using either the "Z" and "z" buttons or the "ZX" and "ZY" buttons at the same location. The appearance of the plot can be changed in many ways. As a simple example, go to the menu "plot", then "set appearance". Here, you can remove the line connecting the individual points (set "line type" to "none"), because it is annoying, and replace the points by some symbols (*e.g.* set "symbol type" to "circle"), which looks nicer. Note that changes will only become effective then you click "apply". Try to play around a little with the items in the "plot" menu. Sometimes you have to click on the "draw" button in the left panel to make your changes visible. You can also do data transformations and fitting under the menu "data" then "transformation". For example, in there, you can select "regression" to get the best fitting of your data to a straight line, and discover that the correlation coefficient (value from 0 to 1 telling if the data fits well to the line) is 0.93 (not too bad[92]).

To save the graph (*i.e.* the full session including all the adjustments you made), go to the menu "file", then "save as" and save your graph under the name `m_vs_m.agr`. To get a printed version in a file for what is currently on your screen, go to the menu "file", then "print setup". Select "postscript" and "print to file", and give the filename `m_vs_m.ps`. Click "accept". Note that this will not yet print, it has just recorded your preferences. To actually print, go to the menu "file", then "print". This will generate a postscript file `m_vs_m.ps`. Often, the portable document format (`pdf`) is more conveninet (to send graphs to your assistant, you should use the latter format). To do the conversion, go to the terminal window and type `ps2pdf m_vs_m.ps`. This will automatically convert the `ps` file to a `pdf` file with the same basis name, in this case `m_vs_m.pdf`. To see the final result, open this `pdf` file *e.g.* with `evince m_vs_m.pdf`.

1.1.16.4 LibreOffice

The package `LibreOffice` is a `UNIX` version of the Windows Microsoft Office, including programs analog to "word", "powerpoint" and "excel". This package is installed in the computers of our computer rooms (see here[93] how to start them). The corresponding file formats are also (at least to some extent) compatible with the microsoft versions.

1.1.17 Printing

To print a document from HCP G24 or HIT F21, first open it with the corresponding application. For instance, `evince` (in the GUI, it is called "document viewer") will work for `pdf` or `ps` documents, `gedit` will work for text files, `firefox` for web pages, or `xmgrace` for graphics. Then,

[91] Like for `gedit`, you can also start the program with the command `xmgrace file &`. This will directly load the indicated file. This file can be either a data file (like `m_vs_m.dat` here), or an `xmgrace` file (usual extension `.agr`) saved at a previous `xmgrace` session.

[92] Using the slope and intercept of the line, you can also predict that the year America will elect an 80 years old Miss, there might well be close to 80 murders by steam, hot vapors and hot objects in the US (this is probably the reason why they preferentially elect them relatively young).

[93] Click on "activities" in the top left corner of the screen. If they are not yet in your "favorite applications", select "show applications" (or press the `<WINDOWS>` key on the keyboard), and type "libreoffice" in the search bar

select the "print" option from the "file" menu. A dialog window appears. Select the printer called "cardstud". This dialog window allows you to change the print settings if necessary. When you are ready to print, click on the "print" button. Then go to the printer[94] and place your ETH student card at the appropriate spot on the printer. Confirm that you want to print by following the instructions on the printer screen. If you don't want to print directly to the printer but to a file, the principle is the same. You just have to select the option "print to file" from the menu of the specific application.

And a little trick. In case you have multiple `pdf` files that you want to combine (append) into a single `pdf` file, you can use the command `pdfunite part1.pdf part2.pdf total.pdf`. to merge the `pdf` files `part1.pdf` and `part2.pdf` into a single file called `total.pdf` (of course, you can also combine more than two files).

1.2 Week 2

This was a lot of material to learn in one go. By performing the tasks given in this section, you will gain some more practice, and at the same time allow the assistant to assess your level. Recall that, depending on your available time and extent of motivation, you may decide to stop at task A, to do both task A and task B, or even to do the full set of task A-task C. We nevertheless recommend you to do at least task B in addition to task A, since what you will train there will be very useful for the next exercises. After each task you complete, you will have to send an e-mail to your assistant with your solution as an attachment. Note that even if you do not complete all three tasks, you should still read the remaining ones and at least give some thought on how you would solve them.

1.2.1 Task A: Mastering the basic `UNIX` commands

In this task, you will have to perform one after the other the operations listed below using the appropriate `UNIX` commands (along with the right options and arguments). Watch out that you will have to write in your report what operations you carried out, so keep track of them[95]. Here we go...

1. Change your current directory to `~/ex1`.
2. Make a copy of the directory `~/ex1P/ex1A` including its contents in the current directory (under the same name `ex1A`).
3. Change your current directory to the newly created directory `ex1A`.
4. Print to screen the absolute path of the current directory.
5. List to screen the contents of the current directory.
6. Create a new subdirectory `not_elements`.
7. Change your current directory to the subdirectory `elements`.

[94] For HCP G24, the printer is in the room itself. For HIT F21, the closest printer is in the physics library, one floor down (be silent when you go there, as it is a library!).

[95] To keep track of them, you can write them on a piece of paper and later copy them in a text file using `gedit`. Smarter is to directly open `gedit`, and right after you executed a command and it worked properly, you copy-paste it into the edited text file. Even smarter, you can solve the exercise and, immediately after, use `history > cmd.txt`. This will print a list of all the commands you executed recently to the file `cmd.txt`. Then you can edit and polish the file using `gedit` to make it final.

8. Move all the files that do not correspond to an element[96] from the current directory to the directory `not_elements` (that we just created before, one level up from the current directory).

9. Change your current directory back to `~/ex1/ex1A` (one level up from the current directory)

10. Make a list of the contents of the subdirectory `elements` and send the list to a file `list.txt`

11. Make a copy of `list.txt` that is called `list_backup.txt`

12. Append the line "these were really elements" to `list_backup.txt`

13. Compare the files `list.txt` and `list_backup.txt`, and send the information on the difference to a file `tmp1.txt`

14. Select from the file `list.txt` all the lines that contain the upper-case letter "R", sort this alphabetically, and send the result to a file `elements.txt`

15. Count the number of lines of `elements.txt`, and append the result at the rear of the file `tmp1.txt`

16. Change the permissions of the file `elements.txt` so that it becomes read only for the user (*i.e.* that you only keep read and execute, but not write permissions[97]).

17. Delete the file `list.txt`

18. Make a list of the directory `not_elements`, replace everywhere the text "ele" by the text "nonsense", and append the result at the rear of the file `tmp1.txt`

19. Delete the directory `not_elements` including its content.

20. Make a detailed list (*i.e.* file-permission modes, last-modification date/time, *etc...*) of the contents of the current directory sorted by last-modification date from the most ancient to the most recent, and send the list to a file `tmp2.txt`

21. Concatenate `line.txt`, `elements.txt`, `tmp1.txt`, `tmp2.txt` and (again) `line.txt` into a new file `ex1A.txt`

22. Make a list (using the appropriate wildcards) of all the files in the subdirectory `elements` that have a name starting with either "S" or "T", followed by exactly one character, and followed by ".ele", and append it to the file `ex1A.txt`

Now, use `gedit` on the file `ex1A.txt`. Above the output (current contents of `ex1A.txt`), insert a header with your personal information, as described in Sec. 0.8 of Ex. 0, the *starting document for students*. In the header, you can omit "program description" (you did not write any program this time), but under "short report", insert a list of the commands you used to perform the operations 1-22 (for the ease of correction, please, report the number 1, 2, 3, ..., 21, 22 at the start of each line). Send the file `ex1A.txt` as an e-mail attachment to your assistant and that's it. You are done with task A.

[96] To help you, there is something called "the periodic table of the elements" that you can easily find on the web. Also: the files themselves contain the name and the atomic number.

[97] As explained in the last footnote of Sec. 1.1.11.3, with the current computer setup, this is all you can change in terms of permissions in your home directory.

1.2.2 Task B: Making a clean graph using `xmgrace`

In this task, you will have to use the program `xmgrace` to make a clean and clear graph comparing the melting and boiling points of *n*-alkanes as a function of the number of carbon atoms they contain.

For this, change your current directory to ∼`/ex1`, make a copy of the directory ∼`/ex1P/ex1B` (including its content) in the current directory, and change your current directory to the newly created directory `ex1B`. In there, you will find the files `mp.dat` and `bp.dat` listing the melting and boiling points, respectively, of the linear alkanes containing from 1 to 12 carbon atoms. In both files, the first column is the name, the second is the number of carbon atoms, the third is the formula, the fourth is the melting or boiling point (value in °C, and corresponding to a pressure of 1 bar). Have a look at the files using `cat`.

The first thing we need to do is to create a file `ex1B.dat` containing in its first column the number of carbon atoms, in its second column the melting point, and in its third column the boiling point[98]. You should be able to do this with `UNIX`. If you feel lost, have a look at Secs. 1.1.12.2 and 1.1.12.9, which contain filters capable of assembling two files column-wise, and printing only specific columns in the resulting file. And if you really (*really*!) don't find out, the solution is in the Appendix (Sec. 1.A.3) - but there will be a price to pay...

Now, type `xmgrace &` and open the file `ex1B.dat` (menu: "data", then "import", then "ASCII"; important: change "load as" to NXY, to specify that we will have more than one curve). Alternatively, you can do this directly by typing `xmgrace -nxy ex1B.dat &` (where the `-nxy` does the above "load as" change). Then click "AS" for auto-scaling. This is already nice, but it is only a raw graph. If you want to include it in a scientific article (or even simply to send it to your assistant), you need to cosmetize it a bit. This is exactly what you will do step-by-step here, following the rough instructions below.

1. Go to the menu: "plot", then "axis properties". Change the axes start/stop so that the x-axis ranges from 0 to 13 and the y-axis from -200 to +250°C. Change the tick spacing so that we have major ticks every 1 unit along the x-axis and every 100°C along the y-axis, and so that we have no minor tick long the x-axis and minor ticks[99] evey 10°C along the y-axis. For both axes, add an axis label that says what is the quantity and what is its unit.

2. Go to the menu: "plot", then "set appearance". For the two curves (which are referred to in the program as S0 and S1), set the symbol/size/lineshape/color. We want the melting point with a blue line and triangle-down symbols, and the boiling point with a red line and triangle-up symbols. Add a string describing what are the two quantities (this will make a legend box appear on the graph).

3. Go to the menu: "plot", then "graph appearance". Give a title to your graph. As a subtitle, add your first name + last name + the timing information (*i.e.* how much time needed to complete this task B). Here, you can also move a bit the legend box, so that it does not overlap with the curves.

4. Go to the menu: "file", then "save as" and save your graph under the name `ex1B.agr` (in case you want to modify it further later).

[98] Alternatively, we could make two separate files containing in their first columns the number of carbon atoms and in their second columns either the melting or the boiling point. Then you have to call `xmgrace` with the names of the two files as arguments. But let's proceed this time with the single-file method.

[99] Watch out that `xmgrace` asks you for the number of minor ticks between two major ticks. So, you have to ask for 9 intermediate ticks and not 10!

5. Go to the menu: "file", then "print setup". Select "postscript" and "print to file" and give the filename `ex1B.ps`. Click "accept".

6. Go to the menu: "file", then "print".

7. In the terminal window, type `ps2pdf ex1B.ps`. This will convert the `ps` file to a `pdf`. Open your `pdf` file with `evince ex1B.pdf` and make sure you are happy with your Meisterwerk.

Send the file `ex1B.pdf` as an e-mail attachment to your assistant and that's it. You are done with task B.

Well, not completely: don't forget to look also at the results with a chemist's eye. Can you rationalize the trends observed in the graph along the n-alkane series? And do the temperatures match your expectations concerning the most stable phase of these alkanes at room temperature and atmospheric pressure? Would the curves go up or down if we considered a lower pressure instead, and which of the curves would be most sensitive to this change? If you care about all this (you should) and have already given it a thought, you can find the answers on the net!

1.2.3 Task C: Advanced text-file processing using `UNIX`

If you have the ambition to (once) kill a balrog and (maybe) become Gandalf-the-White, *i.e.* a real hacker, this last task C is definitely for you. Its goal is to challenge you with *real problem-solving* using `UNIX`. And you will even end up writing your very first `UNIX` shell script. To help you on the way, we will give you many hints. But you will still have to fight hard for the solution. It is particularly important here to remember that the `man` pages of `UNIX` and the internet (search engine) are powerful allies. Ready?

Then, change your current directory to `~/ex1`, make a copy of the directory `~/ex1P/ex1C` (including its content) in the current directory, and change your current directory to the newly created directory `ex1C`.

The directory contains a file `bible.txt`, which is a plain text version of the King James Bible. In this file, each line is a verse of the Bible, starting by the reference of the verse in terms of book, chapter number and verse number (*e.g.* `Gen|1|1|`), and followed by its text. In this directory, you will also discover three weapons that will greatly help you in your quest[100]:

1. The command `del_ref` is a filter that eliminates the verse reference (try `cat bible.txt | ./del_ref | less` and see for yourself).

2. The command `repl_spec` is a filter that will replace all non-alphanumeric characters (*i.e.* anything that is not a letter or a digit) in a text by spaces. It will also add one space at the beginning and at the end of each line (try `cat bible.txt | ./repl_spec | less` and see for yourself).

3. The command `split_line` is a filter that breaks all lines into separate words (one on each line), where the different words are assumed to be separated by spaces or newlines (try `cat bible.txt | ./split_line | less` and see for yourself).

The solution of nearly all (actually, all but one) of the tasks below can be obtained by a *single pipeline*, starting with a `cat`, going through standard `UNIX` filters that we saw in Sec. 1.1.12 (with the appropriate options) or/and the three filters we gave you above, and returning the desired

[100] Don't dream about swords, shields and magic spells. What we give you is far more powerful: `UNIX` filters (*i.e.* commands that do something on their standard input and return the result to their standard output). If you `cat` the files, you will see that these filters are actually very simple `UNIX` shell scripts relying on the commands `sed` and `awk`.

result. A good tip to construct these pipelines is to do it step by step. Let's give a little example: say you want to know how many different books there are in the Bible. To get started, try `cat bible.txt | less` and ask yourself: how do I extract the book name? A good idea is to first kill the `|` characters, so the book name becomes a separate first column. So, you try[101] `cat bible.txt | ./repl_spec | less` and ask yourself: how do I now only keep the first column? Here you immediately (of course!) think about `awk` and try `cat bible.txt | ./repl_spec | awk '{ print $1 }' | less`. Now, we need to pick only one copy only of the repeated adjacent lines with the same book name, so you try `cat bible.txt | ./repl_spec | awk '{ print $1 }' | uniq | less`. You are almost there. Our answer is the number of lines of this. And the pipeline you need is thus `cat bible.txt | ./repl_spec | awk '{ print $1 }' | uniq | wc -l`, which tells you there are 66 books in the King James bible.

Now you can start the tasks. Like in task A, keep track of what you do to solve the problems. At the very end, we will assemble all the commands you used into a shell script, and it is what you will return to your assistant as a solution. Recall that it is possible to solve all the problems below with one single pipeline, except for task 3, where you will need three successive `UNIX` commands.

1. Just to warm up: how many verses are there in the King James Bible?

2. With so many verses, it is probable that there are some repeats... What is the verse that is repeated the largest number of times in the text? Hints: (*i*) before checking for repeats, you need to eliminate the reference of the verse at the start of the line; (*ii*) then, you may want to consider `sort` and `uniq`, with the appropriate options.

3. The file `bible_scrambled.txt` contains the same lines as `bible.txt`, but in random order[102]. In addition, one spurious verse has been added, as you can see by comparing the line counts of the two files. Can you find out what is the added verse? Hints: (*i*) you may want to use `diff`; (*ii*) but `diff bible.txt bible_scrambled.txt` will not be very useful in this case (just try), so you need to first transform the two files in some way; (*iii*) here, the easy solution involves three successive `UNIX` commands.

4. And what if we asked you to solve again the previous task, but you are *not allowed* to use `diff`? Hint: (*i*) maybe it is a good idea to start by concatenating the two files; (*ii*) and then ask yourself about which lines are repeated and which ones are not.

5. As a next task, you will have to find out *in how many verses* the word "God" occurs at least once. We do not care about the spelling, *i.e.* "god", "God" and "GOD" should all qualify. But we do not accept words like "goddess", "godly", "godliness" and the like. Hint: if you manage to kill the punctuation and add a space at the start/end of each line, this will make sure that the occurrences of "God" you want to find are surrounded by spaces on the left and on the right (that might help as a first step).

6. As a next task, you will have to find *out how many times in total* the word "God" occurs in the text, with the same restrictions as above on what we consider to be a valid occurrence.

7. And as a last task, we would like to know in how many distinct spellings of the word "God" occurs, like "god", "God" or "GOD" would make three - but maybe there are others?

Now we are going to assemble this material into your very first shell script. Open a new file `ex1C.csh` using `gedit`. Write the first line `#!/bin/csh`, which will tell `UNIX` that what is in the

[101] To save time, remember that you can recall, modify and re-execute the last command line using the key "arrow up", Sec. 1.1.15.

[102] There is a cool and (nearly) useless `-R` option to `sort` that does exactly this.

file is a c-shell script. Then write a header with your personal information, as described in Sec. 0.8 of Ex. 0, the *starting document for students*. Watch out that the text should not be surrounded by `/* ... */`, which corresponds to a comment in `C++`. Instead, use a `#` at the start of each line, which is the way comments are indicated in a shell script. In the header, you can omit "program description" and "short report". Then write the commands you used to solve tasks 1-7 of the exercise above just as you typed them in the terminal window. Precede each command by some comment on the way it works, and an `echo` command stating what will be printed. For example, for task 1 and the start of task 2, this should read

```
#
# For task 1, I just use 'wc -l', which reports the number of lines in a text file
#
echo "1. The number of lines in the King James Bible is:"
cat bible.txt | wc -l
#
# For task 2, I ...
#
echo "2. The most repeated verse and the number of times it is repeated in the Bible are:"
...
```

Finish your script by the command[103] `exit`. Now give yourself execute permission[104] for `ex1C.csh` Now type `./ex1C.csh` and enjoy your success!

For sending the file to your assistant, first rename it from `ex1C.csh` to `ex1C.txt` (this is because, for security reasons, the ETH *outlook* e-mail system blocks attached files with a name ending in `.csh`). Then send `ex1C.txt` as an e-mail attachment to your assistant and that's it. You are done with task C.

At this point, sit back and enjoy, because you are done with `ex1`. Make sure you have a relaxing week-end, because next week is an important one in your life: we start *programming*!

1.A Appendix

This appendix contains supplementary material for the curious ones. It does not belong to the main material of the exercise (and the scope of the final exam).

1.A.1 Additional information on command path and aliases

Here, you will find some supplementary information pertaining to Sec. 1.1.9.4.

By default, the path variable (*i.e.* the list of directories where `UNIX` looks for commands) does not include your current directory. As a result, if you have an executable `my_program` in your current directory, you can run it as `./my_program`, but using simply `my_program` won't work (error message: "command not found"). If you like, you can fix this. For this, go to your home directory and edit a (hidden) file called `.cshrc` using `gedit`. Be careful not change or delete anything in this file! Just go to the end of the file, and add `setenv PATH .:${PATH}` as a last line. Save the file. Then type `source ~/.cshrc`. And that's it. Now `UNIX` looks for commands in your current directory too, and typing just `my_program` will work. Note that directories are searched in the order specified by the path variable. So, if your current directory contains a file named `ls`, it is this one that will be executed when you type `ls`, and no longer the standard `ls` command. If later you write many own programs, the standard way would be to make a directory

[103] This is optional, but cleaner.

[104] As explained in Sec. 1.1.11.3, this is actually not necessary for the present system, where the user always has read and execute permission. But it would be needed on a proper `UNIX` system.

~/bin for them, and use setenv PATH .:/home/user/bin:${PATH} (where user is, as usual, your own nethz user name). Now UNIX will search first in the current directory, then in your ~/bin directory, and then in the standard directories.

1.A.2 Additional information on using the network

Here, you will find some supplementary information pertaining to Sec. 1.1.14.

If you frequently use the network to hop from one computer to another (*e.g.* between your desktop at home, your laptop, and the ETH computers), here are a few tricks to simplify your life. First, you may become annoyed to type everytime csh after you login (or open a new terminal window), see Sec. 1.1.4. This is needed because your default shell is bash and not csh (and it is really nicer to work with csh!). To make csh your default shell, you just have to create a file ~/.bash_profile (as the name starts with a dot, this is a hidden file, that you can only see with ls -a) containing the single line csh. You may also have to create a file ~/.bashrc with the same content. That's it: now, when you login or open a new terminal window, you no longer need to type the csh. Note: in some cases, this may cause login problems when you use some remote applications on your computer (this has already happened, but I am not entirely sure when). In this case, just do rm~/.bash* and go on the manual way.

1.A.3 How to make the file ex1B.dat

To make the file ex1B.dat, the command is paste mp.dat bp.dat | awk '{ print $2,$4,$8 }' > ex1B.dat. The price to pay for this tip is that we unfortunately cannot congratulate you on figuring out the answer yourself (yes, I know, it's tough!).